A FRAMEWORK FOR INTEGRATING NESTED QUERIES IN NATURAL LANGUAGE INTERFACES TO DATABASES

Synopsis submitted to CHRIST (Deemed to be University)

as partial requirement for the award of the Degree of

DOCTOR OF PHILOSOPHY

IN

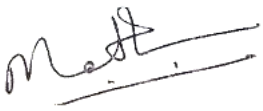
COMPUTER SCIENCE

by

MATHEW GEORGE

Under the Supervision of
Rohini V
Professor

CHRIST
(DEEMED TO BE UNIVERSITY)
BANGALORE · INDIA



Signature of Scholar Signature of Supervisor(s)

CENTRE FOR RESEARCH
CHRIST (Deemed to be University)
BENGALURU

FEBRUARY 2022

CENTRE FOR RESEARCH

Approval of Synopsis

[as per clause 11(k.) of CU Regulation for the Degree of Doctor of Philosophy, May 2019]

Name of Scholar:	Mathew George
Department:	Computer Science
Name of Co-Supervisor:	Dr Rohini V
Names of Research Advisory Committee: 1. Dr. Beaulah Soundarabai P. 2. Dr. Senthilnathan T. 3. Dr. Manjunatha Hiremath	
Name and Designation of External Reviewer if applicable:	
Date of Synopsis Presentation:	Dec-2021
Title of the PhD thesis:	A Framework for Integrating Nested Queries in Natural Language Interfaces to Databases

Approved by:

1. Name of Supervisor: Signature:
2. Name of RAC Member: Signature:
3. Name of RAC Member: Signature:
4. Name of External reviewer if applicable Signature:

Head of Department

Date of approval by Doctoral Committee:

Director Centre for Research

TABLE OF CONTENTS

1. INTRODUCTION 2
2. NEED FOR THE STUDY 3
3. LITERATURE REVIEW 3
4. PROBLEM STATEMENT 7
5. OBJECTIVE OF THE STUDY 7
   5.1 Primary Objective 7
   5.2 Secondary Objective 7
6. THE FRAMEWORK DESIGN 7
   6.1 The Key Ingredients of the Translation Framework 8
   6.2 Translation Steps 8
   5.3 Detailed Translation Steps 8
7. METHODOLOGY 9
8. PREPARATORY CONCEPTS 9
   8.1 Domain Ontology 9
   8.2 Sketch Synthesis 9
   8.3 Fault Localization and Program Repair 9
   8.4 Employing Rewrite Methods 10
9. THE ARCHITECTURE 10
   9.1 Overview 11
   9.2 System Architecture 12
   9.3 Assumptions 12
   9.4 The Translation Process 12
   9.5 Algorithm 13
10. THE INTERMEDIATE REPRESENTATION 14
   10.1 Overview 14
   10.2 The Query building Process 16
   10.3 Intermediate Language modeling 19
   10.4 Intermediate Language modeling with Functions 22
   10.5 Lambda Calculus 22
   10.6 Lambda Calculus and Combinators 25
   10.7 Fixed-point Combinator 25
   10.8 A Translation Scenario 25
11. ADVANCED METHODS IN SYNTHESIZING NESTED QUERIES 26
   11.1 Key Concepts 27
   11.2 Advanced Concepts used in Translation 27
   11.3 Structural Recursion 27
   11.4 Lambda Calculus combined with Comprehensions 28
   11.5 Monad Comprehensions 29
   11.6 Sketch Generation 29
12. RESULTS 31
   12.1 Results Discussion 34
13. CONCLUSIONS 35
14. FUTURE WORK 36
   REFERENCES 36
   List of Tables 43
   List of Figures 43
   Conferences and Publications 44

ABSTRACT

To translate Natural Language (NL) statements into Structured Query Language (SQL) queries, different methods and systems were proposed in the past. This work presents a framework for automating the translation of Data Requirement Specifications (DRS) given by enterprise business users in NL into Structured Query Language (SQL) queries, focusing on requirements that result in the generation of nested SQLs. The framework takes the business user's DRS given in English as input and generates an initial query sketch by employing semantic parsing. This initial sketch is further refined and completed into a well-formed SQL by consulting the Database Schema. It performs the translation by combining NL processing techniques with Query Sketch generation methods and refines it further by employing Repair techniques. The framework suggests Lambda expressions for intermediate representations and leverages the operations of Lambda Calculus for performing the transformations needed in translation. Though Lambda Calculus is found to be effective in representing the intermediate expressions and helps in performing the transformations that are needed for translating specific predicates into SQL, its inflexibility in combining parallel computations is a constraint. This work recommends the use of advanced language constructs for performing the required transformation at the intermediate language level. To represent clauses that are in parallel or are in pipeline, and to perform the required transformations on the intermediate expressions involving these, more advanced programming constructs are needed. It also adopts functional programming techniques to deal with complex scenarios involving nested queries. This work recommends the use of advanced language constructs (like Fixed-point Combinators and Monad Comprehensions) for performing the required transformation at the intermediate language level and adopts functional programming techniques (For eg., fold functions) as syntactic sugar. The importance of Context Switching and rendering it as a repair technique in the automatic translation of NL statements into Structured Query Language (SQL) statements is also investigated. Lambda Context Calculus (LCC) provides the operational semantics and the relevant methods needed in transforming NL statements into SQL, preserving the integrity and compositionality[5] of expressions in every step of the translation. Rewrite techniques assist greatly in handling computations based on Contexts and play an important role when a translation is performed in multiple steps. Integrating Context Switching techniques along with other known techniques of translation like Semantic Parsing, Repair by Rewrite differentiates this work from the rest.

1. INTRODUCTION

Program Synthesis refers to the automatic creation of programs in a Domain Specific Language (DSL) capturing user intent by satisfying a given set of constraints expressed as formal DRS. This work presents an end-to-end framework for synthesizing SQL queries from NL statements in the school of NL Interface to Databases (NLIDB) systems. The techniques specified in building this framework are implemented as a system named REX (abbreviation of **Re**search **Ex**pert), which eventually synthesizes SQL queries from DRS. Typically, Rex is an automated, database-aware system with limited dependency on the user's knowledge of the underlying database schema.

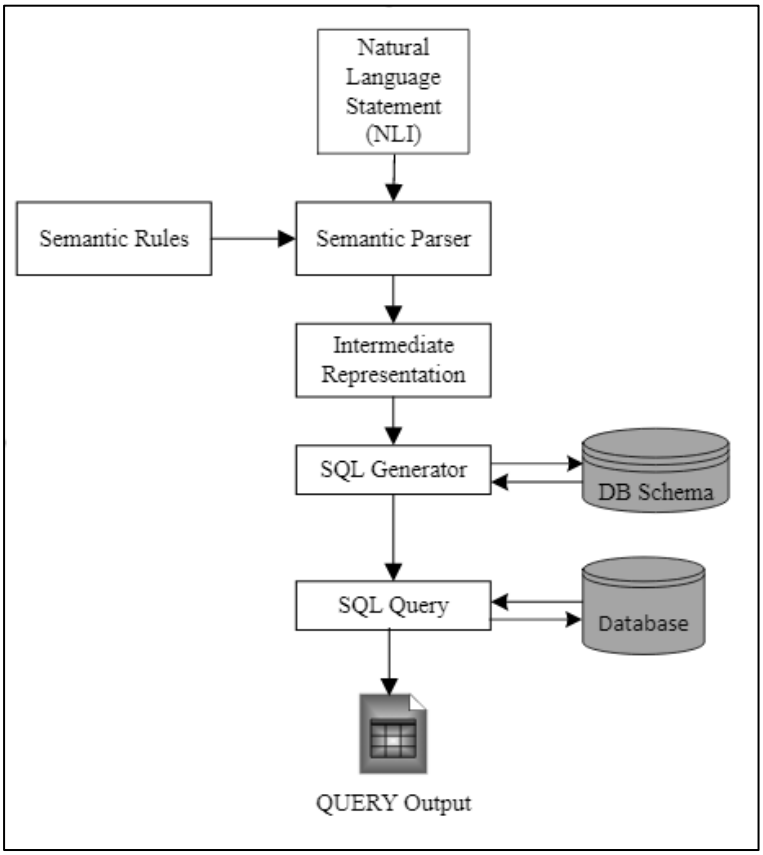


Fig 1: System Components

For translating DRS to SQL, after generating an initial sketch by way of Semantic Parsing, the framework employs various transformation techniques such as sketch parsing, refines it further to generate intermediate code, to form the final SQL query. Though NLIDB systems are usually implemented by adopting logic-based approaches operating on an intermediate representation, a domain ontology-based systematic application of operational semantics is a unique characteristic of this work making it novel and the transformation a dynamic, well-guided automated process. Direct translation from NL statements into well-formed SQL queries is tedious. Developing packages to aid in automatic Machine translation too is

cumbersome. The primary advantage we see in the intermediate representation is that it offers an intermediate platform for implementing the inference mechanisms that utilize the database schema information integrating domain ontologies.

This dissertation will also show how program synthesis benefits from functional programming concepts developed by functional programming communities. As it is common in any Natural Language Processing (NLP) system, the approach starts with Semantic Parsing Techniques and proceeds to combine Functional Programming techniques thus making NL translation for SQL program synthesis essentially a functional programming activity. Hence relevant Functional Programming Techniques must be invariably applied as explained in this work for completing the nested SQLs creation process.

2. NEED FOR THE STUDY

Business users and SQL developers involved in data analysis and conversion activities, often query the data stored in Relational Databases. Large Re-engineering and application upgrade projects rely on SQL developers for extracting data from the source database to load it into target tables. Automatic creation of nested sub-queries and their integration into the main SQL trunk is an area that has accomplished less success due to its incremental complexity.

The proposed framework takes the business user's DRS given in English as input and eventually generates a well-formed SQL by introducing a few novel methods and combining them with existing techniques in the NL translation and program synthesis domains resulting in the creation of an end-to-end Natural Language Interface to Databases (NLIDB) framework. This work also demonstrates how program synthesis benefits from functional programming concepts developed by functional programming communities.

3. LITERATURE REVIEW

Literature Review has been primarily focused around NLIDB Systems, Semantic Parsing, Functional Programming Techniques, Lambda Calculus and Machine Learning. Earlier efforts by Navid and Dillig [1] resulted in a system called SQLizer, in which the user writes a program sketch containing holes (placeholders to be replaced with appropriate expressions), and holes in query sketches were completed using predicates containing tables, columns, or expressions containing these. Even though the initial substitution is an add predicate it, in turn, can contain satellite predicates, allowing further extension. As in SQLizer, REX's SQL synthesis is primarily decomposed into separate tasks like a) Sketch generation b) Sketch Repair and Extension. In addition to sketching, this work borrows concepts from λ-Calculus [34] and adopts Function Abstraction Techniques [2] for blending the required abstractions and

compositionality[5] required in the intermediate language representation and subsequent transformation.

Earlier, algebraic approaches dominated the intermediate language representation of query structures and predicates used in translation. SQL is chiefly designed from abstractions given by relational algebra. SQL is primarily based on bags, while relational algebra operates on sets [2]. M. J. and Lengrand, S [3] describe the proofs and reduction rules associated with Lambda Calculus and manage the complexity of Lambda Context Calculus in a simple and modular way.

SKETCH system (Program Synthesis by Sketching), proposed by Solar-Lezama [31], is leading work in the domain of program synthesis frameworks. It takes as input an initial sketch or partial program, generated by semantic parsing of the input text. It is typically a program template or plan with holes in place of desired sub-expressions to be substituted later by Domain-Specific Search (schema probing in the case of SQL synthesis).

ORAKEL [63] is an ontology-based Natural Language Interface (NLI) for querying a knowledge base and relies on a deduction process to answer a user's query, devised an inference engine to provide an answer to the user's query. IBM researchers, Saha and et-al [53], discuss a model that enables users to query SAP-ERP data using NL. Their paper, "Natural Language Querying in SAP-ERP Platform", presents the effectiveness of the natural language query interface on a set of questions given by a set of SAP practitioners.

FERRY [43] is a new system that translates a first-order nested query with a nested result into multiple flat SQL queries (though a fixed number of queries based on the type), expanding on Van den Bussche [44] and translating the queries to SQL. In standard SQL, many recursive queries are mentioned as not expressible [37]. The FERRY [43] system is known to give a total translation, but without supporting abstraction [37].

Keyword-based systems: Keyword-based systems assist non-experts to formulate ad-hoc queries using a set of keywords from an intentional set of English statements. Samples of earlier work in this area can be found in systems such as BANKS [4], DISCOVER [14], and DBxplorer [40]. Later works in this area [5, 36, and 33] supports aggregate functions [36] or logical connectives in predicates [33]. In a graph (tree) based representation, listing all entities (nodes) that comply with some specified first-order requirement, can adopt set operations to perform aggregate and other specific operations (like grouping). In general, NLIDB systems support a range of versatile sets of queries than systems based on keywords.

Natural language interface to databases (NLIDB): NLIDBs have been at the forefront of translation interfaces for decades altogether [45]. Unlike other database query interfaces such as keyword, template, input-output example, or graph-based ones, NLIDBs allow users without much programming skills to formulate more complex SQL queries. Earlier works in this domain focused on developing question-answer-based systems that are bound to specific databases. Later works describe NLIDB systems suitable for multiple databases but required customizations [32, 43, and 35]. That means, primarily these techniques were not database-independent but additional customizations made the trick in adapting to each database. More recent NLIDB systems such as PRECISE [27] and NALIR [19], aims to be database-independent. The PRECISE system is developed based on the assumption that there exists a correspondence between certain classes of predicates and their corresponding SQL so that it is always possible to accurately map the predicate into its specific SQL. Based on this insight, Popescu et al. [26] [27] consider a class of semantically traceable DRS and describe a system for formally translating such queries to SQL [30, 29].

Many systems are not database-independent (e.g., [43]) and hence need to be trained on a specific database for their respective translation techniques to work. Other techniques e.g., PRECIS [30], NALIR [19], and NaLIX [20]) aims to work on a broader class of databases but suffer from other shortcomings. For instance, the PRECISE system [27] works only for a subset of NL predicates, and NALIR [19] depends on the user for guidance, to choose the right target table and column names to build the query structure, to be effective. Metadata-based approaches have been actively pursued quite recently [73, 74]. The techniques we describe in this paper do not rely on database-specific training but consult meta-data and hence are schema-aware.

Query synthesis from other means refers to synthesizing database queries from informal specifications other than NL. Here, users convey their intent by way of input-output examples, and the approaches developed in [18, 11, 12], operate in a limited or bound context of the database schema.

Another approach to query synthesis is from code written in popular programming languages [10, 9, and 7]. For instance, Query by Synthesis (QBS) [7], converts a range of imperative constructs in the application logic into relational specifications and then to SQL queries. This approach is not meant for end-users and can be deemed as a form of code refactoring.

Programming by natural language: As SQL is a declarative language; built upon relational algebra, the approach of this paper has some similarity to programming by NL[13, 8, 29, 28, 5] and uses denotational semantics [8]. NL has also been used for creating smartphone automation scripts [5], the “If-This-Then-That Recipes” [28] that demonstrates how to

effectively map language to code, in spreadsheet programming [13], and string manipulation [29] based on ontology mapping, facilitating interoperability between the source and target domains of transformation. Among these, the NLyze tool [13], though it targets spreadsheets rather than relational databases, combines semantic parsing with type-directed synthesis, is most closely related to SQLizer. However, semantic parsing alone has pretty low recall and hence needs to be augmented by combining other techniques like type-directed synthesis and ARS (Abstraction, Reference, and Synthesis) based systems.

Complete functional languages were used as an intermediate query language in query compilation and optimization efforts [56, 57]. This same concept can be effectively used in DRS to SQL translation as we also need a type-based intermediate language that maps the entities and relationships preserving the equational nature of SQL predicates. λ-calculus is identified as an ideal formalism as it was originally developed to study mathematical properties of effectively computable functions and has provided a strong theoretical foundation for the family of functional programming languages. Moreover, it presupposes the possibility of forming complex queries by the functional composition of higher-order combinators[2]. The functional view makes query synthesis subject to the extensive collection of program transformation techniques—as in Bird-Meertens' formalism, or as in Squiggol [48]. A content-sensitive Neural Text-to-SQL generator is proposed in TypeSQL [61] which views the problem as a slot filling task and uses type information to better understand rare entities, predicates, and numbers in the input. A sketch-based two-step neural Dual-Encoder model known as the SEQ2SEQ model [24], that combines syntactic elements with semantic information is proposed lately.

Paper	**Year**	**Authors**	**Limitation/Research Gap**
Type-and Content-Driven Synthesis of SQL Queries from Natural Language	2017	Yaghmazadeh, Navid and Wang, Yuepeng and Dillig, Isil and Dillig, Thomas	The research does not detail how nested clauses can be effectively constructed
Natural Language Interface for Databases Using a Dual-Encoder Model.	2018	Hosu I., Iacob R., Brad F., Ruseti, S. and Rebedea T.	This neural encoder model does not mention how nested Sub-Queries can be handled.
How to comprehend queries functionally	1999	T. Grust and M. H. Scholl	This work deals with Translating SQLs embedded in programming languages into Object Code and not from Natural Language to SQL Automation.
An End-to-end Neural Natural Language Interface for Databases.	2018	P. Utama & et-al,	A future avenue of development is to allow users to build **Complex Nested Queries** incrementally.

Natural Language Interface for Databases Using a Dual-Encoder Model	2018	I. Hosu et al,	Does not mention how **nested Sub-Queries** can be handled.

Table 1: Comparative list of top SQL automation efforts

4. PROBLEM STATEMENT

Develop a Framework for synthesizing nested SQL statements from Data Requirement Specifications (DRS).

5. OBJECTIVE OF THE STUDY

5.1 Primary Objective

Develop a formal Framework for creating Nested SQL statements from Data Requirement Statements given in NL.

5.2 Secondary Objective

Develop an Intermediate (Logical-Form) Representation or Pivot Coding to formalize the NL to SQL Translation.

6. RESEARCH DESIGN

Over the course of the DRS to SQL translation, after generating an initial sketch by way of Semantic Parsing, the framework employs various transformation techniques such as logical-form (sketch) parsing, refines it further to generate a more compact intermediate code, to form the final SQL query by applying relevant Transformation methods and techniques. The design is primarily guided to handle the C4 level of SQL complexity as given below without undermining lesser complexity levels C1-C3.

SQL Complexity	Description
C1	No Aggregate functions or JOIN operators
C2	No Aggregate function but JOINs different tables
C3	Uses an Aggregate function
C4	**Uses Sub-Query or Self-join**

Table 2: SQLs by Complexity

6.1 The Key Ingredients of the Translation Framework

Here are the Key Ingredients of the Translation Framework:

- A Graphical User Interface (GUI)-based Interface or Template
- Semantic Parser (Use any standard parser)
- Intermediate Representation (Logical-Form)
- Transformation Methods and Techniques
- Develop the Proof of Concept by integrating Transformation steps into a pipeline

6.2 Translation Sequence

As the initial sketch need not bring us the anticipated results in the first go by the underlying database schema, the approach subsequently performs a type-and-effect check and repairs the incorrect parts of the sketch. The current work extends the automated fault localization and repair tactics that are used to refine query sketches [1].

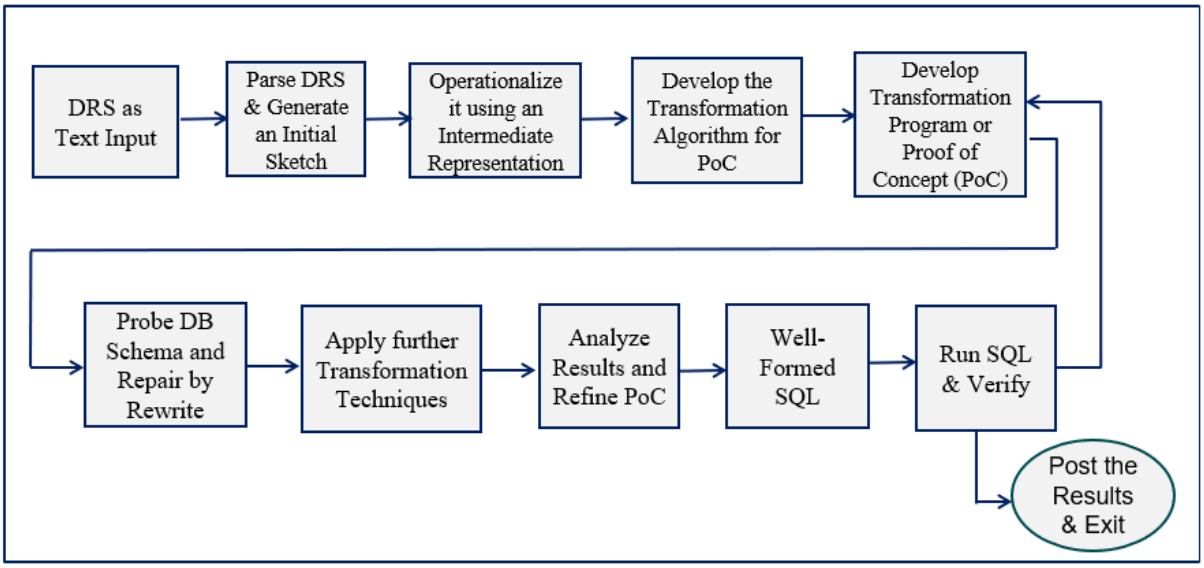

DRS as Text Input
Parse DRS & Generate an Initial Sketch
Operationalize it using an Intermediate Representation
Develop the Transformation Algorithm for PoC
Develop Transformation Program or Proof of Concept (PoC)
Probe DB Schema and Repair by Rewrite
Apply further Transformation Techniques
Analyze Results and Refine PoC
Well-Formed SQL
Run SQL & Verify
Post the Results & Exit

Fig 5: Framework Diagram

6.4 Detailed Translation Steps

Listing the Steps of Translation as below:

- Parse the Input Text (DRS)
- Identify Composite (Parent) Entities in the Sketch.
- Create the Initial Sketch using Denotational Semantics and Lambda Calculus
- Probe the DB Schema by Path Finding (Hierarchical Tree Traversing).
- Perform Bifurcations (create new/bifurcated entities by consulting DB schema) and perform Substitutions, if applicable.
- Apply Lambda Calculus Operations for Sketch Transformation
- Repair by Rewrite
- Apply Context Switching for Repair
- Develop the Transformation Program to form a well-formed SQL
- Use Functional Programming Techniques for Optimisation
- Pass these Entities into a programming pipeline to synthesize Nested SQLs.

7. METHODOLOGY

To create SQLs from DRS, this work prepares an initial sketch using abstract clauses in an intermediate language, then rewrite it as connected intermediate expressions and subsequently translate it syntactically and semantically into SQL. The initial sketch is repaired using repair techniques, refined and extended further by type and schema-driven query synthesis and perfected by applying transformation techniques. For generating SQLs from DRS, it has become necessary to design and implement an intermediate representation (pivot coding) resulting in the creation of an intermediate Language (Abstract Query Language) that expresses and translates natural language descriptions into query skeletons that eventually gets translated into SQL. It also uses abstract functions to represent nested relations and to perform operations on these functions in the translation process. It relies on the type and schema-driven synthesis methodology [1] to generate SQL queries from query sketches and extends it further for nested queries.

8. PREPARATORY CONCEPTS

8.1 Domain Ontology

The entities, their properties, and inter-relationships correspond to the domain ontology. Ontology-based systems offer richer semantic support as it directly consults database schema and provide a sound basis for translation.

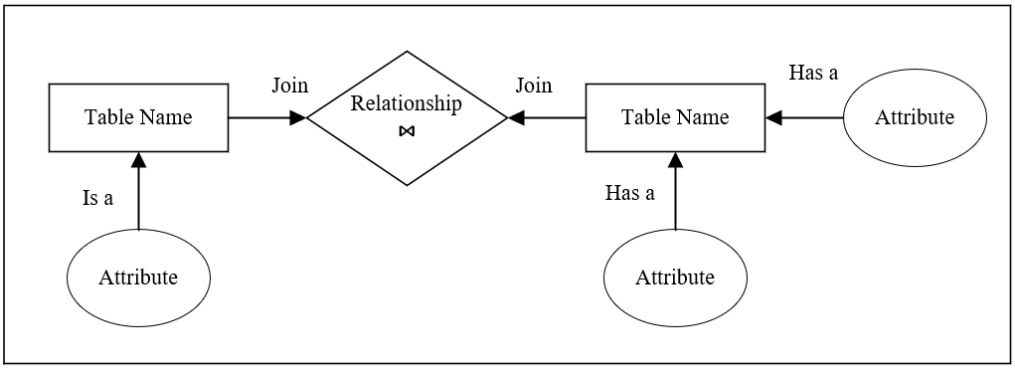


Fig 2: Database Domain Ontology

8.2 Sketch synthesis

Sketching offers the first level of abstraction and enables complex algorithms to be initiated with minimal effort so that the initial Sketch is subsequently repaired through the automation program that can be probe underlying schema and manage the data structures involved.

8.3 Fault localization and program repair

The fault localization and program repair are inspired by recent works on these topics [15, 22, 23, 39, and [27]. Fault localization techniques are employed to identify the expression that

caused the SQL to fail [15]. Like program repair techniques the fault localization techniques, tries to pinpoint the smallest problematic subpart of the initial expression or "program". This work performs fault localization at the intermediate representation itself by finding the term that evaluates as false.

8.4 Employing Rewrite methods

Rewrite systems provide a standard method for modeling computation [34]. DRS in NL is first converted into logical forms, which are transformed into well-formed executable statements and applied to a knowledge base to obtain result sets. Once we have translated NL statements into formal logic, we can draw logical inferences or perform transformations based on the domain knowledge from which the formal expressions derive their meanings. To create SQLs from DRS, this work prepares an initial sketch using abstract clauses in an intermediate language, then rewrites it as connected intermediate expressions and subsequently translates it syntactically and semantically into SQL. The initial sketch is repaired using repair techniques, refined, extended further by type and schema-driven query synthesis, and perfected by applying transformation techniques.

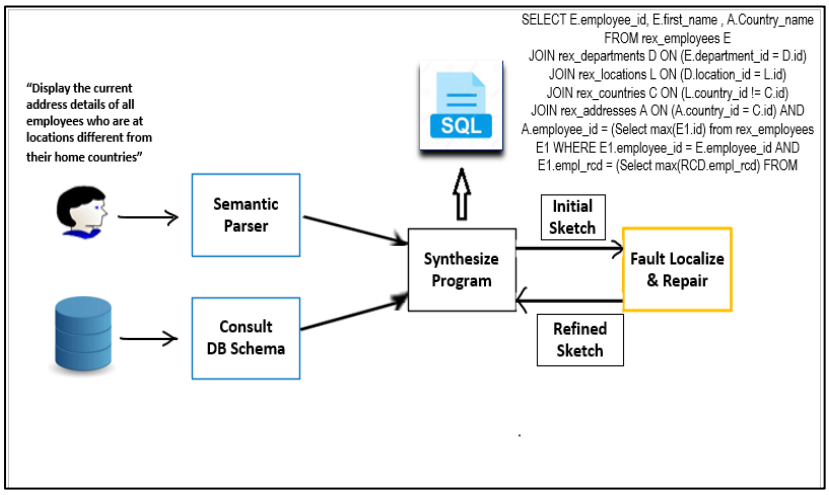

"Display the current address details of all employees who are at locations different from their home countries"
SELECT E.employee_id, E.first_name , A.Country_name
FROM rex_employees E
JOIN rex_departments D ON (E.department_id = D.id)
JOIN rex_locations L ON (D.location_id = L.id)
JOIN rex_countries C ON (L.country_id != C.id)
JOIN rex_addresses A ON (A.country_id = C.id) AND
A.employee_id = (Select max(E1.id) from rex_employees
E1 WHERE E1.employee_id = E.employee_id AND
E1.empl_rcd = (Select max(RCD.empl_rcd) FROM
SQL
Semantic Parser
Consult DB Schema
Synthesize Program
Initial Sketch
Refined Sketch
Fault Localize & Repair

Fig 3: Schematic overview of the approach

An overview of the framework is schematically shown in Figure 3 with a refinement loop extension that incorporates each of the three key techniques (Sketch Synthesis, Fault localization, and Repair) as outlined above.

9. THE ARCHITECTURE

9.1 Overview

The aspects of a program synthesis problem are broadly categorized into below three main streams and finally integrated through the REX Framework.

- Intent specification

- Program space
- Search strategy

The program space is initially represented by intermediate logical expressions (using a variant of Relational Algebra and Lambda Calculus) and later guided by SQL syntax for synthesizing a well-formed structured query statement, and the search strategy is based on deduction.

Another unique characteristic of this work is the amount of initial effort spent in axiomatizing the application domain into a reasonably complete system of deductive rules. Several notational conventions have been adopted to represent the end-users specification given in text form and transform it before reaching a stage where functional programming techniques can be applied.

The following are the unique characteristics:

1) Proposes a three-stage translation approach that
    i. First translates an input DRS into an intermediate representation defined using an intermediate language construct(s) and
    ii. Perform schema probing to get additional information apart from the initial inference made from the user's intent expressed in the DRS and
    iii. Apply operational semantics to translate the intermediate representation (Sketch) into its desired SQL query.
2) An algorithm to translate the NL statements or DRS into SQL queries. The algorithm can handle various scenarios based on the underlying schema orientation.
3) The system is made more dynamic with Context Switching methods for repairing and reconstructing the SQL if the initial attempt(s) fails.

The following Predicate logic expressions will help us to understand the work better.

Terms: Refers to entities. Represented as: Variables: x, y, z | Constants: 'John', 'Peter'
Functions: applied to terms: FatherOf('John')
Predicates: Refers to properties of or relations between entities Eg: tall'(x), eat'(x,y),
Formulas: Evaluates to True or False
Atomic formulas: Predicates, applied to terms: tall ('John')
Complex formulas: Constructed recursively via logical connectives and quantifiers

The **key technical contributions** discussed in this section are:

1. How Lambda expressions can be effectively used to represent predicate clauses, including nested ones. We often use λ-expressions to construct complex logical formulas which adapt well here too.

2. How operations of Lambda Calculus can be applied effectively for refining the initial sketch.

3. Use of Fixed-point Combinator is proposed for function abstraction while translating repeated nested expressions from the Abstract Query Language to an equivalent nested Sub-query.

9.2 System Architecture

The overall architecture of the Rex NLIDB interface is shown in Fig4 below.

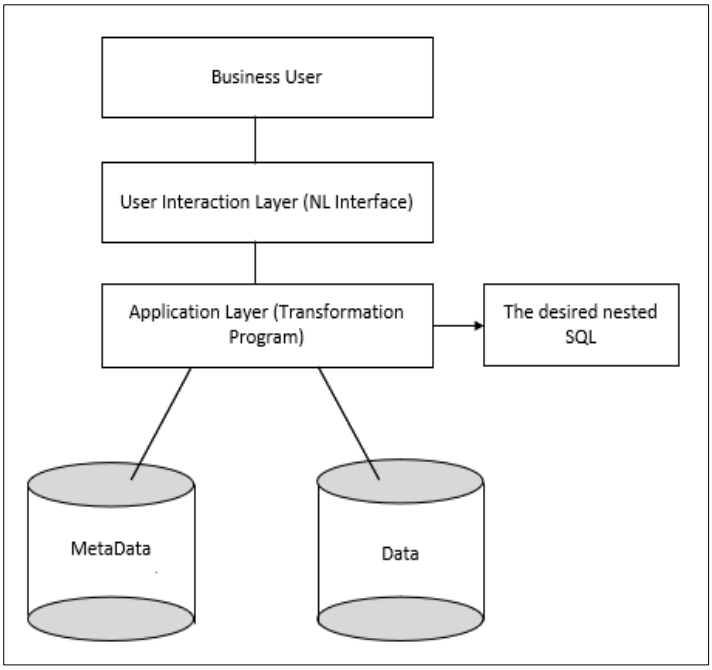


Fig 4: High-Level Overview of the System

9.3 Assumptions

1) As extensive research and development have gone into building Semantic Parsers in the last 50 years or so, this work will not be concentrating on building a parser; instead, any reasonably good (say Stanford NL parser) will be used.

2) Though this work presents a framework for synthesizing nested queries from NL statements; the focus is on translating cases that result in nested SQLs.

3) There exists a certain amount of correspondence between the domain of users' NL statements (with Enterprise Entities and their relations represented as Hierarchical trees) and the SQL statement environments (Database Entities and their relations represented by Schema).

4) The work is not for trivial users but for business users who have some knowledge of the Database Schema.

9.4 The Translation Process

REX, like the ORAKEL [63] relies on a well-defined deduction process to answer a query, whose main requirement is that the user's question is translated first into a logical form and represented, to eventually translate it into a SQL query by systematically

applying transformation techniques integrated into a synthesis program engine (also known as the Proof of Concept (POC) in the research context).

Please find below a sample Schema probing SQL (for the Emergency_Contacts of the Employee) and the results returned:

```
SELECT tc.table_name, kcu.column_name, ccu.table_name AS foreign_table_name,
ccu.column_name AS foreign_column_name
FROM information_schema.table_constraints AS tc
  JOIN information_schema.key_column_usage AS kcu ON tc.constraint_name =
kcu.constraint_name
  JOIN information_schema.constraint_column_usage AS ccu ON ccu.constraint_name =
tc.constraint_name
WHERE constraint_type = 'FOREIGN KEY'
and tc.constraint_name like '%rex_%' and ccu.table_name not like '%res_%'
and tc.table_name like 'rex_emergency_contacts';
```

	table_name character varying	column_name character varying	foreign_table_name character varying	foreign_column_name character varying
1	rex emergency contacts	employee id	rex employees	id
2	rex emergency contacts	contact id	rex contacts	id

Fig 6: Sample Output of the Schema Probing SQL

9.5 Algorithm

1) Input DRS
2) The Semantic parser will identify and return the key entities from source to target
3) Probe the schema using these entities and get real table names
4) Create an Initial Sketch as a well-formed SQL
5) Run the Initial Sketch and get the records
6) If len(returned_records) == expected_rows then return the SQL and exit
7) Else perform fault_localize in the SQL and call Repair Procedure
8) Repair by ***substitution***
9) Repair by ***bifurcation***
10) Repair by SQL extension by adding predicate clauses, use DB schema probing SQL
11) For constructing predicate clauses, use ***Fixed Point Combinators*** and DB schema probing SQL
12) For routing of logic in predicate clauses use ***Monad Comprehensions*** and DB schema probing SQL
13) For optimization and integrating multiple clauses to the query trunk use the ***foldr*** feature and DB schema probing SQL
14) Run the Final Sketch and verify the records
15) If len (returned_records) == expected_rows then return the final SQL and exit
16) Else perform **context switching** to repair the current SQL by traversing the entities/nodes backward
17) Repeat the Repair Synthesize process until the synthesized SQL retunes the expected records

The algorithm bifurcates (or rather splits) the nodes that have the potential for more than one arc, and at times reverses the direction (and performs context Switching) as few arcs are only viable.

10. THE INTERMEDIATE REPRESENTATION

10.1 Overview

The choice of an adequate intermediate representation is a major step in the overall translation and repair process. For creating nested SQLs, we need an intermediate programming paradigm that has the semantic simplicity of relational algebra and the expressive power of functional programming languages. Hence the special emphasis is kept on the intermediate language representation and application of required transformation techniques in the ambit of a complete translation framework. The much-preferred type-based design - based on initial algebras[4] of a core functional language is followed and intermediate representations that suit the demands of nested query creation are subsequently developed. Advanced type systems are needed in the design of an intermediate language for representing nested Queries. Applying relational query processing rules alone will not be sufficient for these extended-type systems. Moreover, types are to be dynamic as they are automatically inferred during the translation process. All these deliberations point to the fact that a combination of approaches involving structural, logical, and domain ontology-based heuristic semantic processing is required for brokering the mapping between intermediate representations and target SQL forms.

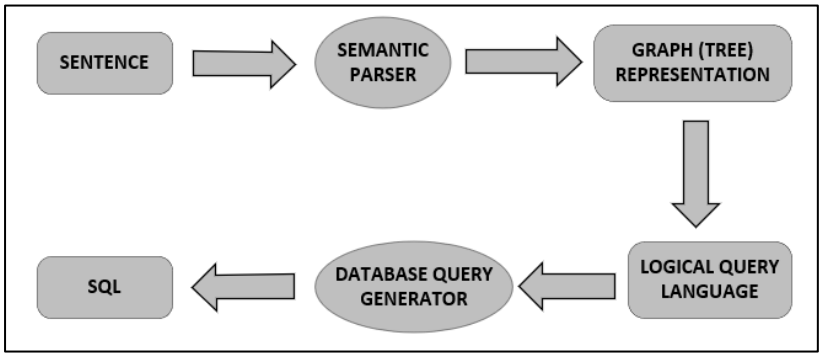


Fig 7: Transformation Flow

The formalization methods adopted in this work are founded on the principle of compositionality - the representation of any statement is a function of the representations of the components of the statement. The two major distinguishing characteristics of this research are:

a) Comprehending queries as functional programs and
b) Adoption of functional components as the most viable transformation paradigm for forming complex queries [2].

10.1.1 Programming Language Semantics

One of the effective means to define a programming paradigm or language is as an executable formal specification. This at least has two advantages:

(1) enables ***execution of programs*** and

(2) can ***formally verify or reason*** about programs and programming paradigms.

10.1.2 Graphical Representation and Translation

For visualization and modeling purposes, we create Graph (mostly Hierarchical Tree) based representations corresponding to all the selected sets of entities from the DRS and subsequently draw interpretations using their properties and relationships. Graph-based schema representations offer intuitive conceptualization and reduce translation complexity at the intermediate level of transformation typically for domains like ERP databases where data organization is hierarchical. Though the Graph database model is usually hidden in the Schema definition of ERP databases and not explicitly visible, it can be discovered by traversing through the meta-data tables. The meta-data traversing can employ techniques like breadth or depth-first approaches or use algorithms for a path or rather shortest pathfinding, least-cost path traversal and finding, methods for discovering cliques or dense sub-graphs, identifying strongly and weakly connected entities, etc. A labeled and directed property graph will be a suitable model for use here whose basic elements include:

• Entities as nodes
• Properties as attributes
• Labels as types
• Relationships as edges having a direction, starting from a source node, and ending at a destination node
• Identifiers

Conceptual modeling can start by viewing attributes as empirical typed functions originally described as predicates in the DRS. Multiple (relational) attributes can result in the construction of several graph sketch variants. The hierarchical representation can reduce the functional Complexity. Simple Integrity Constraints can be represented as a Functional Dependency (FD) and the rest as a Conditional Functional Dependency (CFD). The *home_address of an employee* is an example of FD but *the latest effective dated current address is an example of CFD*. More complex scenarios can be represented using *subgraphs and super-graphs*. Figure 8 shows a schematic representation of the two interpretations as separate tree spans. It is typically identified by the schema probing SQL, bypassing the list of entities and relations returned by the parser for the NLQ given in Figure 5.

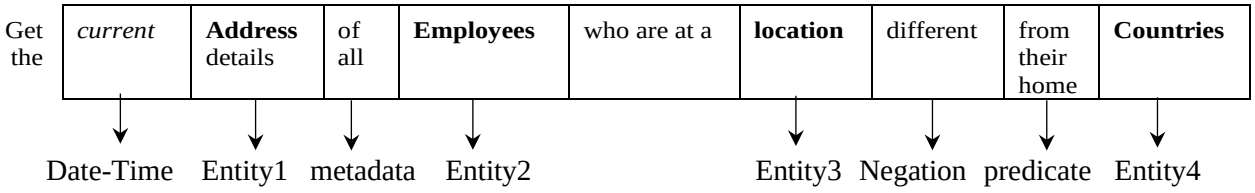


Fig 8: A Natural Language Query (NLQ) annotated for Ontology Mapping

Single-value attributes of the form *key: attribute* is suitable here, where an *attribute* can eventually belong to elementary types like *String, Number, Date,* etc. For example, the unique identification key

of the Employee entity can be #Emp_id. At the DB schema level, the identification key of Employee can be either{Employee_id or Emp_id, id}.

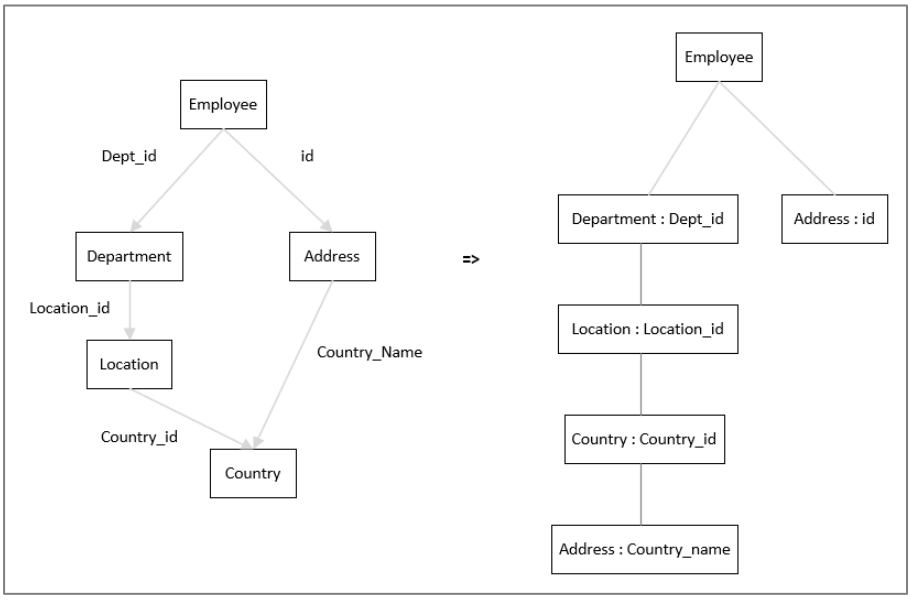


Fig 9: A Graph to Tree Conversion

It offers a suitable ecosystem to deal with relations and graphs in one integrated environment. Graph-based models aid in conceptual and data level representation of entities eventually facilitating the functional modelling of entity relationships.

10.2 The Query building Process

The translation algorithm operates on each intermediate expression, performing attribute and JOIN clause transformations until all the unknown elements of the intermediate query are rightly substituted. For an unknown attribute, if the current property is not identified as a field in the database table of the current context, the attribute transformation operations (assisted by Schema probing SQL) will proceed to deduce the property from another context that is related to the current context through an inheritance (parent-child) relationship or a membership relationship. A membership change will trigger the replacement of the current query with a new query or a set of new queries, typical to the new member context or contexts. Also, whenever the tables that were suggested initially cannot be joined directly, the physical table or tables in the join path need to be probed and the original join conditions will get replaced. The transformation may need a source entity bifurcation operation for getting the new additional table or set of tables involved, creating new join conditions on the fly.

10.2.1 The special cases of nested Queries

In the intermediate representation, a Sub-Query is placed in the where clause as a subQueryExpr expression. The inner query and the outer query are essentially connected by a property belonging to the outer query. If the inner query has a predicate applied on

aggregation over the property of the outer query, the inner query is accommodated in the having clause of the outer query.

```
SELECT E.Employee_id, max(E.empl_rcd) as empl_rcd, COUNT(E.Employee_id)
FROM rex_employees E
GROUP BY E.id
HAVING max(E.id) = (SELECT max(A.Employee_id) FROM rex_addresses A WHERE A.Employee_id=E.id);
```

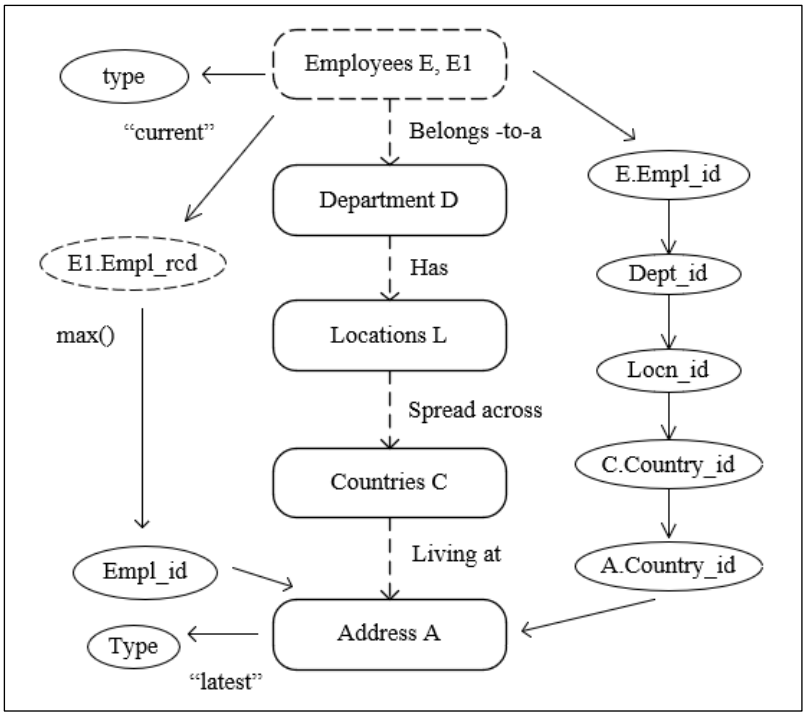


Fig 10: Graphical depiction of the top-ranked interpretations of the NLQ given in Fig8.

10.2.2 Significance of Relational Algebra-based notations

A Lambda Term is a variable, an abstraction, an application, or a pair but not the attribute of a selection, hence relational algebra terms too are needed to represent selection and projection. The grammar in Figure 10 (an adaptation of relational algebra) will serve as a base to formalize the query sketch generation and synthesis process.

Projection: **Π**

Selection: **σ**

Join: ⋈

Physical tables: **t** (or views created in the database)

Column names: ***c***

Variables: ***v***

$\chi := \Pi_L(\chi) \mid \sigma_\varphi(\chi) \mid \chi\ c \bowtie_c \chi \mid t$

$L := L, L \mid c \mid f\ (c) \mid g\ (f\ (c), c)$

$E = T \mid c \mid v$

η := χ | *?[]* | *v*

ϕ := ϕ *lop* ϕ | ¬ϕ | *c op E*

op := ≤ | < | = | > | ≥

lop := ∧ | ∨

ω = *t ON* ϕ

Database schema represented as: **Γ**

A Sketch (or a relation - at a lower level) is represented by: **χ** ***(chi)***

A simple Expression: ***E;*** An unknown expression: ***η***

A hole or an unknown expression (wild card predicate expression) is represented as: *[]*

Specifiers: ψ, κ

Predicate: ϕ

f ∈*AggrFunc* = **{***max, min, avg, sum, count***}** can be represented as a subscript for the projection Π, for instance, $\Pi_{f(c)}(T)$ returns a single aggregate value obtained by applying function f to column c of relation T, For example, $\Pi_{max(emp\_id)}(T)$.

Fig 11: Grammar of Extended Relational Algebra. Here χ at its simplest level represents a relation and t table name; *f* denotes an aggregate function, and *v* denotes a value.

The relations, represented as χ in the grammar, are produced by applying the relational algebra operators, Π, σ and ⋈, include physical tables-T or views created in the database. Projection $\Pi_L(\chi)$ takes a relation χ and a column list L and returns a new relation that only contains the columns in L. The selection operation $\sigma_\varphi(\chi)$ yields a new relation that only contains rows satisfying φ in χ. The JOIN operation $\chi_{1\ c1} \bowtie_{c2} \chi_2$ composes two relations χ_1, χ_2 such that the result contains exactly those rows of $\chi_1 \times \chi_2$ satisfying c1 = c2, where c1, c2 are columns in χ_1, χ_2 respectively.

10.2.3 Verification

If there are no rows returned by relation *T* satisfying predicate Φ, then $\sigma_\Phi(T)$ has a low probability of occurring in the target query.

10.2.4 Handling JOINs and Aggregate functions

To simplify the technical representation, the table name or its alias names is prefixed to each column name. The present work only considers equijoin for the initial sketch creation because they are the most commonly used join operator and can later extend it to other kinds of join operators (e.g., θ-join). However, the grammar from Figure 10 has the potential to represent

nested queries at a high level through the expression of the form ***c op E***. For instance, joins can occur within other joins or predicates represented by φ.

When a function is applied to a set of values residing in a table or relation to return a resultant value of more significant meaning (as defined by the function) is known as aggregation. Aggregations can be of two types: typical aggregate functions (such as count, sum, average, min & max) and grouping functions (GROUP BY). The Relational Algebra variant is shown in Fig2 also accommodates aggregate functions and group-by operators. For conciseness, aggregate functions ($f \in AggrFunc$) = {*max, min, avg, sum, count,* etc.} are specified as a subscript in the projection operation. Specifically, $\Pi_{g(f(c1),c2)}(\chi)$ divides rows of χ into groups g_i based on values stored in column c2 and, for each g_i, it yields the aggregate value f(c1).

10.3 Intermediate Language modeling

10.3.1 Language of Terms (LT)

A variant of typed Lambda Calculus that can be used to query graph representation of databases is the Language of Terms. It is a language defined by Pokorny [78] for querying Graph Databases (GD).

10.3.2 LT Types [42]

1. Let *'T'* be the set {BOOL, NUM, STRING}. (*When referring to types in T, we will ignore typecase and will use 't')*

2. Then the set of all types **T** (\mathbb{T}) is defined as:

$\{(t_1 \times \_\_\_ \times t_n)\ \forall_i \in \{1, \ldots., n\}\ t_i \in \mathbf{T} \cup \{t_x!\ t_y \mid t_x, t_y \in \mathbf{T}\} \cup T$ {node

For an edge label vocabulary E, the node property keys and their respective types are represented by N and the scope of variables and their types as S. Then x is a term with type y in scope if and only if (x; y) ∈ term S

Constants

c, (c, NUM) ∈ Term(S) | The numeric value of c.

c, (c, BOOL) ∈ Term(S) | The true/false value of c.

c, (c, STRING) ∈ Term(S) | The string value of c.

Terms of Primitive Types

(s, NUM) ∈ Term(S) for s ∈ R

(TRUE, BOOL) ∈ Term(S)

(FALSE, BOOL) ∈ Term(S)

(s, STRING) ∈ Term(S) for s an ASCII String

Terms of comparator function types

∀A ∈ (T \ BOOL) (=A, (A×A) → BOOL)) ∈ Term(S)

$\forall A, B \in (T \setminus BOOL)\ (>A, (A \times A) \rightarrow BOOL))$

$\forall A, B \in (T \setminus BOOL) \in Term(S)$

$\forall A, B \in (T \setminus BOOL)\ (<A, (A \times A) \rightarrow BOOL)) \in Term(S)$

$\forall A, B \in (T \setminus BOOL)\ (>=A, (A \times A) \rightarrow BOOL)) \in Term(S)$

$\forall A, B \in (T \setminus BOOL)\ (<=A, (A \times A) \rightarrow BOOL)) \in Term(S)$

Terms of logical function types

$(\exists, (node \rightarrow BOOL) \rightarrow BOOL) \in Term(S)$

$(and, (BOOL \times BOOL) \rightarrow BOOL)) \in Term(S)$

$(or, (BOOL \times BOOL) \rightarrow BOOL)) \in Term(S)$

$(!, BOOL \rightarrow BOOL) \in Term(S)$

The existential quantifier ∃ is a method of testing whether a function results in TRUE for a single element in a set. Also, Projection can be modeled using the existential quantifier function $\exists. ((A \rightarrow BOOL) \times (A \rightarrow BOOL)) \rightarrow BOOL.$

Universal quantifier (*)

For universal quantifiers holds that $\forall(f1, f2) =! \exists (f1, !f2)$.

The universal quantifier (Select *) can be modeled using set difference and the existential quantifier.

10.3.3 Set difference

Set difference concerning entities (neds in graph model) can be expressed as a combination of union and negation. However, while querying, negation is a kind of set difference. Set difference for columns can be modeled with other building blocks like !=.

Set difference using LT

Let q_1 and q_2 be Queries with type $(T_1 x \ldots x T_n) \rightarrow BOOL$.

Then q1 \ q2 can be queried with.

$\lambda x_1{:}T_1, \ldots, x_n{:}T_n(q1(x_1, \ldots, x_n)$ and $!q_2(x_1, \ldots, x_n))$

Cartesian product using LT

Cartesian product non-strictly increases the number of columns in a result set.

Let q_1 and q_2 be queries with respective types $(T_1 \times \ldots \times T_n) \rightarrow BOOL$ and $(T'_1 \times \ldots \times T'_n) \rightarrow BOOL$.

Then the cartesian product of q_1 and q_2 is:

$\lambda x_1{:}T_1, \ldots, x_n{:}T_n, y_1{:}T'_1, \ldots, y_m{:}T'_m\ (and(q_1(x_1, \ldots, x_n), q_2(y_1, \ldots, y_m))$

Terms of lambda function types

$\forall (f, t_f) \in Term\ S \cup \{(x1, t1), \ldots, (x_n, t_n)\})\ \forall i \in \{1, \ldots, n\},$

negation $\exists y \in \mathbf{T}\ (x_i, y) \in S\ (\lambda x_1{:}t_1, \ldots x_n{:}t_n(f), (t_1 \times \ldots \times t_n) \rightarrow t_f) \in Term(S)$

$\forall$ (f, ($t_1 \times \cdots \times t_n$) $\rightarrow$ s) $\in$ Term(S)

$\forall i \in \{1 \times \cdots \times n\}$, ($x_1$, t_1) $\in$ Term(S)

(f (x_1, ..., x_n), s) $\in$ Term(S)

Application of terms of function types

$\forall$ (f, ($t_1 \times \cdots \times t_n$)! s) $\in$ Term(S) $\forall_i$ v {1, … , n}, (x_1, t_1) $\in$ Term(S)

(f(x_1,..., x_n), s) $\in$ Term(S)

This means that a function of type T$\rightarrow$S that is applied to appropriate argument(s) T are of type S.

Functions based on edge connections

$\forall$e $\in$ E (e, (node, node) $\rightarrow$ BOOL) $\in$ Term(S)

Edge labels may be used as function names and defined accordingly to test whether two entities/nodes are related/connected.

Retrieval of node properties

$\forall$ (p, t) $\in$ N $\forall$(n, node) $\in$ Term(S) (n, p, t) $\in$ Term(S)

The dot notation is used to retrieve Node properties are from terms of type node

10.3.4 Selection using Lambda Terms

Selection denotes the selection of a subset of rows of a query result exactly as in relational algebra, where equality or inequality is made from one of the columns of the entities under consideration. Selection however cannot be implemented using aggregation but can be combined with it.

Let q be some query with arguments x_1, ..., x_n of types t_1, . . . , t_n respectively,

Then a selection with $x_i > c$ can be written as.

$\lambda x_1.t_1$, ..., $x_n.t_n$(and(q(x1, ... , x_n),> (x_i, c)))

10.3.5 Projection

In relational algebra, projection is defined as returning the result of a query, mostly with a subset of all available columns, carefully picked by the user. Since the result of a query is a set removing relevant columns may result in duplicate entries or fewer entries. Projection can be modeled using the existential quantifier function $\exists$ or using the ***foldgroup*** function as explained below.

10.3.6 Grouping

Grouping helps to aggregate specific sets of rows in the result set returned by the query expression.

We can use this for example to sum the leaves of each employee from the HRMS database. The query would look like this:

*Foldgroup (**λ**x:num, y:num(x+y), 0, **λ** hidden p:node, e:string; s:num(and*

*(=STRING (p.employee, e),=STRING (p.leave l))), **λ**x:string, l:string(x))*

10.3.7 Set union

Union helps in adding the results of two queries for which the results are of the same type (set of columns). The set union can be derived from the set difference. Let q_1 and q_2 be two queries with result sets s_1 and s_2 respectively, of the same type t such that it has universe U_t. Then $s_1 \cup s_2 = U_t \setminus (U_t \setminus s_1\, s_2)$.

10.3.8 Set intersection

Set intersection though has similarities with a set difference, but is symmetrical, which brings the fundamental difference. As a Set operation, it can be defined using set difference: $q_1 \cap q_2 = G \setminus (G \setminus (q_1 \setminus q_2) \setminus (q_2 \setminus q_1))$.

10.3.9 Node access using Lambda Terms

Current representation of Lambda Terms uses λx, y(f(x, y)) rather than λx:node, y:num(f(x, y)).

Let $q = \lambda x_1$:node, x_2:t_1, . . . , x_n:t_n (F(x_1, . . . , x_n))

This query made of Lambda Terms is defined as the query for which the result returned is the cross product of all lambda variable combinations and possibilities for which F yields TRUE. If the user defines F to be TRUE for some x1, . . . , xn, the result set's first column will return nodes.

10.3.10 Edge access using Lambda Terms

Along with entities (nodes), the relational databases stores relationships too as relational constraints. These can be accessed by Lambda Terms to retrieve nodes as part of an interim selection that can be taken up for further processing in constructing SQLs.

Let *e* be some edge label (so $e \in E$).

To test whether two nodes are connected by an edge with that label, the below (logical) expression can be used/implemented:

$(e, (\text{node} \times \text{node}) \rightarrow \text{BOOL}) \in \text{Term}(S)$

10.4 Intermediate Language modeling with Functions

As the employment of higher-order functions is found to be essential for the NL to SQL translation, Lambda Calculus is adopted in this work at the outset so that the computational transformation and the subsequent methods and techniques used have a sound mathematical basis.

10.5 Lambda Calculus

Lambda Calculus is the universal Assembly Language that any computation can be translated to, making it a useful tool in intermediate language representation and translation. λ-Calculus combines all the necessary ingredients of operational semantics and is also the meta-language for representing formal definitions in Denotational Semantics. It has all the ingredients to be

a prototype programming language. The λ-Calculus syntax is concise and simple. There are notations for variable identifiers, function abstractions, function application, and sub-expressions. The lambda calculus has a formal syntax, capable of making a language of lambda terms, blended by an array of transformation rules, allowing operations of transformations to be performed on lambda terms. The transformation rules can be used for theoretical proofs or as a sequence of transformation steps.

λ-Calculus is used:

- As an abstract model of a programming language rather than a practical programming language.
- To represent the Denotational Semantics of some languages
- As an intermediate language for a compiler [41]
- Using Lambda calculus, the attribute of a table entity can be represented easily and its predicates can be represented by lambda expressions facilitating implementation via an executable typed function [78] in the automation program.
- The operations of Lambda Calculus can be used as a means for manipulating expressions involving functions in an intermediate representation or language.

  The expression $\lambda x.e_1$ means occurrences of x in the body e1 are bound.

  To evaluate (λx.e1) e2 means to evaluate e1 with x bound to e2

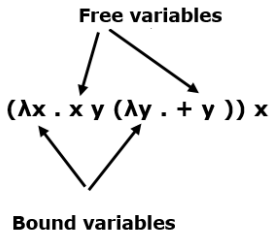


Fig 12: Lambda Expression Representation

9.5.1 Types

The *(elementary) types* $t_1, \ldots, t_k$ $(k \geq 1)$ constitute a base **B**. B consists of both standard types like (Bool, String, Number, etc.) and entity types. Standard types cater to domains of attributes. Their elements can also be constants like, Active =’True’, Country = ’India’, Emp_id = ’1001’, Eff_dt = ‘current date’. **Bool** allows to type entities assets and attributes as relations. In functional programming, they correspond to unary and n-ary functions.

If T, $R_1, \ldots, R_n$ $(n \geq 1)$ are types, then

(T: R1,…,Rn) is a *(functional) type* …. (i)
(R1,…,Rn) is a (tuple) type ….. (ii)

The set of *types* **T** *over* **B** is the least set containing types from **B** and those defied by (i)-(ii). When Ti in **B** is interpreted as non-empty sets, then (T: R1, . . . , Rn) denotes the set of all (total or partial) functions from R1 x…x Rn into S, (R1, . . . , Rn) denotes the Cartesian product R1 x…x Rn.

"the home_address of an employee" is a (country: Employee, address) a (partial) function *f*: Employee x Address → country. More formally, attributes are functions of type ((T:S: W), where W is the logical space (possible worlds), S contains instants, and T belongs to S.

The fact that *X* is an object of type **R**∈**T** can be written as X_R, or "*X* is the **R**-object". For each typed object *o*, the function type returns type(*o)* ∈ **T** of *o*. Logical connectives, quantifiers, and predicates are also typed functions. If T and U, are types then T→ U is also a Type.

10.5.2 Typed Lambda Calculus

Mathematical structures like sets, relations, functions, and assumptions can be represented as Contexts (an expression with an open place or hole) in typed lambda calculus. And examples of operations involving these structures are algebras and relations. Algebras collect sets and perform operations on them satisfying certain properties, and fresh relations can form new sets, that too on top of existing relations and properties governing them.

Suppose x and y are variables and T, S & R lambda terms. The expression T[x:=R] denotes substitution of R for x in T in a capture-avoiding manner. This is defined such that:

(λy.T)[x:=R] = λy.(T[x:=R])

If x ≠ y and y are not in the free variables of R, then the variable y is said to be "fresh" for R.

10.5.3 Substitution

Substitution, written E[V:= R], is the process of replacing all free occurrences of the variable V in the expression E with expression R.

Substitution on terms of the λ-calculus can be accomplished using recursion on the structure of terms, as given below

(λy.M)[x := N] ≡ λy.(M[x := N])

(Where x and y are only variables while M and N are λ expressions). If x ≠ y provided y ∉ FV(N). [For accomplishing substitution in a lambda abstraction, sometimes it is necessary to α-convert the expression. For example, it is incorrect for (λx.y)[y:= x] to result in (λx.x), because the substituted x was supposed to be free but resulted as bound. The proper substitution, in this case, is (λz.x), corresponds to α-equivalence or reduced to α-equivalence].

The meaning of lambda expressions is defined by how expressions can be reduced. There are three kinds of reductions:

***α*-Conversion:** changing bound variables (alpha);

β-Reduction: (Evaluation of an expression by substitution of its variables or terms). The expression to which a redex reduces is called its reduct; the reduct of (λx.M)N is M[x:=N].

η-Conversion: This represents a notion of extensionality (eta).

10.6 Lambda Calculus and Combinators

The expressiveness of a complete functional language is not required to form an intermediate query language. Instead, a small set of combinators suffices to represent queries. Functional languages are usually based on Lambda Calculus defined by mapping to an intermediate language which comes with an extensive equational theory.

10.7 Significance of Fixed-point Combinator [3]

In a type-and-effect system, the types (attributes of relation in this case) tell where sub-queries can be placed and when they must be turned into virtual tables (common-table expressions/views). The typing rules (relationship constraints) permit no recursion in its current context, and thus is analogous to simply typed λ-calculus. Add recursion later using a fixed point combinator [37] for arriving at the next context of tree traversal. A fixed-point function with polymorphic type parameters can aid in context switching.

10.8 A Translation Scenario

This section describes the actual translation process with the help of an example. Figure 4 shows the relevant portion of the schema of an ERP database. Suppose the system wants to synthesize a database query to "Retrieve the ***current*** **address** details of all **employees** who are at a ***location*** different from their home ***countries***". To use our method, the user provides this or a similar English description as DRS. Now, laying out the steps adopted by REX in synthesizing the desired SQL query.

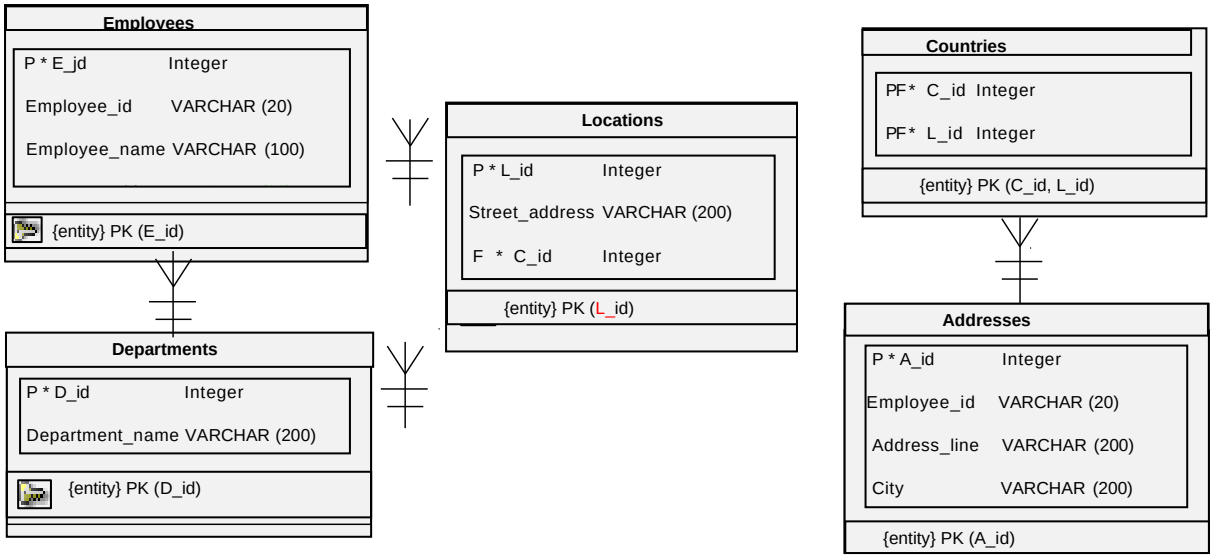


Fig 13: Simplified schema of an HRMS ERP database

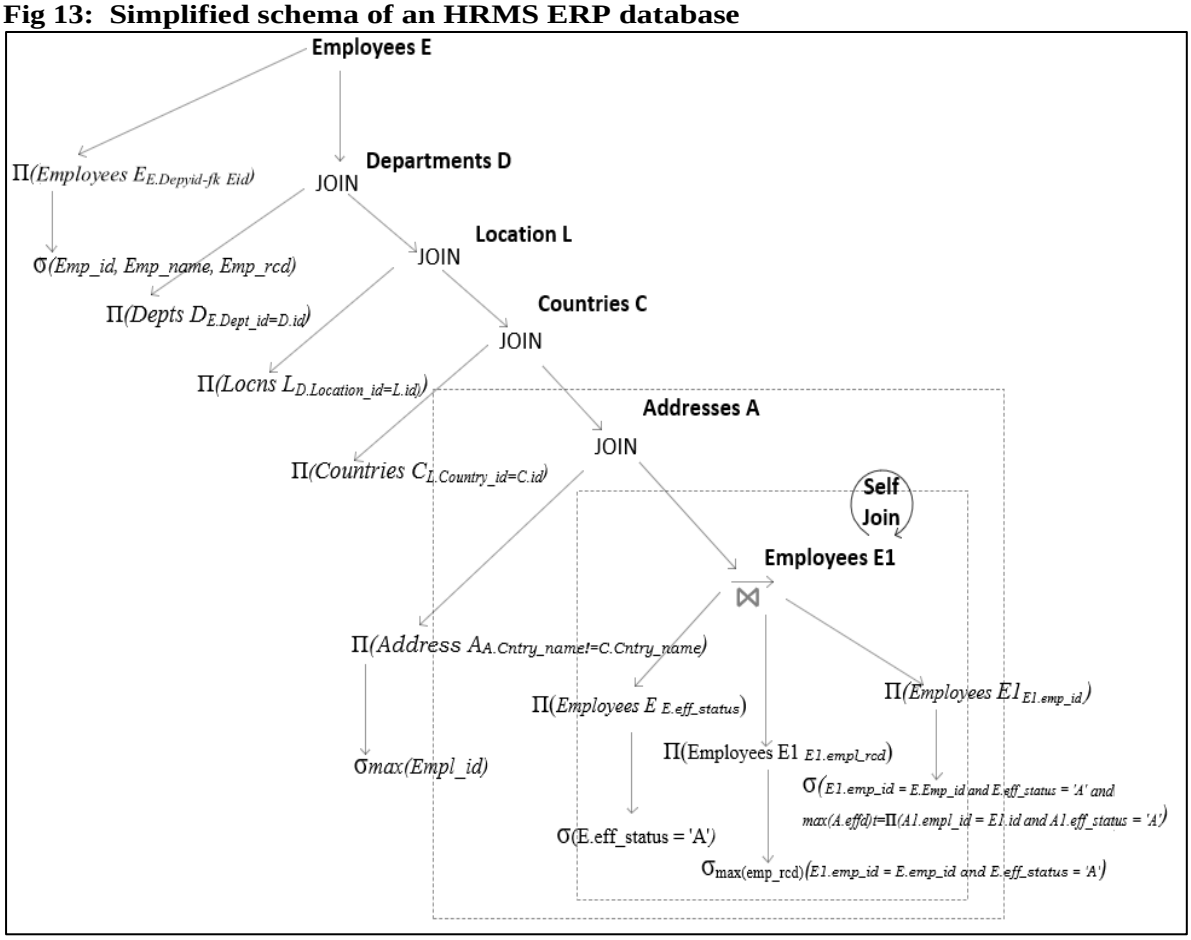


Fig 14: Representation using Relational Algebra

11. ADVANCED METHODS IN SYNTHESIZING NESTED QUERIES

The query algebra operators are an abstract representation of underlying procedures implemented by the query engine. Query predicates were viewed as annotations to algebraic operators and were not part of the translation or compilation phase but were treated later during

the optimization phase. But after representing it in the intermediate language using lambda expressions, they can be transformed by performing Lambda Calculus operations.

11.1 Key Concepts

The key transformation techniques discussed in this paper include

1. How Structural Recursion and its implementation in the form of Comprehensions can be effectively used to represent and implement sub-queries in a pipeline.

2. A more advanced functional programming construct Monad Comprehension is proposed for function abstraction and rallying expressions in a pipeline while translating from the Abstract Query Language into an equivalent SQL query.

3. Application of higher-order combinators like folds (foldr) for abstract representation and as a means to augment recursive processing initiated through fixed-point Combinators.

11.2 Advanced Concepts used in Translation

The tool REX (the query synthesis program) is extended by implementing the advanced techniques discussed here.

[1]***Lambda Calculus*** is a formal system in mathematical logic to express computation based on function abstraction and application by variable binding and substitution. It is a universal model of computation and can simulate any Turing machine.
[2]***Combinator***: A Combinator is a λ-calculus expression to represent primitive functions that have no free variables. A Combinator represents closed expressions (no free variables) of a language and corresponds to axioms of a deductive system.
[3]***Fixed-point Combinator:*** Fixed-point Combinators are used for implementing loops in Lambda calculus. They are also used to implement recursion without calling the function name recursively but by applying the function to itself with a new set of values for its bound variables whenever it is (re)applied.
[4]***Folds:*** In functional programming, folds refer to the use of a given combining operation, recombine the results of recursively processing its constituent parts, building up a return value.
[5]***Compositionality:*** is the notion that a system or an expression should be designed by composing together smaller subsystems or terms, and deliberation about such a system should be done recursively on its structure.

11.3 [6]Structural Recursion

Structural recursion is a restricted form of recursion. Structural recursion makes it possible to express the requirements of iteration, aggregation, and quantification which are at the core of

any database query language and is suitable for relatively complex nested SQLs creation scenarios.

Structural recursion is defined as a top-down, recursive function, much like tree traversing which evaluates the data top-down. Structural recursion is found in almost all tree traversals. Nested queries have a natural correspondence to structural recursion. The DRS to SQL transformation program should take bags of input data, process it, move it out of bags by placing them into sets and manage a flexible type-system across while traversing different levels of the tree. Structural recursion allows the implementation of better algorithms for the same functionality that can be achieved through other programming techniques found around first-class functions.

11.4 Lambda Calculus combined with Comprehensions

Functional languages are usually based on lambda calculus and supported by a solid equational theory that is eventually compiled and interpreted. A complete functional language is not needed to represent queries at an intermediate language level; instead, a small set of combinators would suffice. Theoretically, complex queries can be formed from the functional composition of higher-order combinators. Even though combinators would suffice to represent closed predicates, the sublanguage with combinators needs to be extended with functional programming techniques like comprehensions to combine and pipeline different components into a sequel. Also, while executing, interim results need to be communicated across operators because the functions implementing them are fashioned to take their specific inputs and pass intermediate results.

The combinator sub-language can be extended by applying the syntactic sugar comprehensions which provide a better abstraction of the query intermediate representation. As DRS is more declarative than imperative, adopting Comprehensions have proven a very convenient construct in the creation of SQL kind of declarative query statements [51, 52]. As Comprehensions create data structure from iterators and combine loops and conditional tests in a compact way, they can be employed as an effective intermediary construct while translating DRS to SQL. Just like query languages, comprehensions are provided with variables, variable bindings and allow nesting of predicates arbitrarily without propagating side effects of any predicates involved. Hence comprehensions and basic combinators typically complete the intermediate language [2].

Comprehensions connect related predicates with ease and are predisposed for query predicate transformations. Without the use of comprehensions, this would have needed the application of complicated sets of rewriting rules.

11.5 [7]Monad Comprehensions

Monads provide a framework for bundling/structuring the semantic representation of features such as state, exceptions, and continuations [49]. Monad Comprehensions are recommended for use at the intermediate language level to bundle related components and features, by arraying them in a pipeline to ensure connectivity and continuity between constituent parts. Different types of query nesting correspond to nested representations of monad comprehensions. Monads provide a framework to combine a wide variety of programming paradigms, such as managing state, exceptions, or input-output. It has a return operator that creates values, and a bind operator to link the actions in the pipeline; and its definition follows a set of axioms called monad laws, all these are mandatory for the composition of actions in the pipeline to work properly. The result is the outcome of the entire unit.

While Combinators facilitate the abstraction of query operators and predicates, Monad comprehensions facilitate a calculus-style intermediate language. Calculus sub-expressions with the appropriate combinators are similar to relational calculus but have better expressiveness. The type-based foundation and uniform representation of our intermediate language (IL) allows us to adopt functions (overvalues of an initial algebraic data type τ), and structural recursion constructs like **foldr** provide the fundamental way to combine SQL predicates.

11.6 Sketch Generation (The Case Study extended for the second level of Nesting)

Prepare the SQL for the following DRS statement: “Retrieve the current address details of all employees who are at a location different from their home countries based on their employment records”.

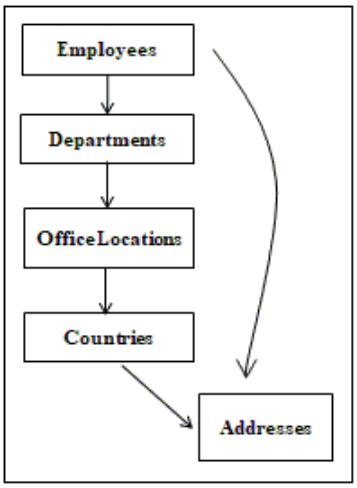


Fig 15: DB Table Relationship

EXAMPLE1: Consider the “Employee” and “Addresses” tables from below, where column names with suffix “_fk" indicate foreign keys.

Eid	Name	Dept_id	Home_Country	Status
101	Mathew	30-Mar-00	India	I
101	Mathew	29-Feb-00	India	A
102	Joe	29-Feb-00	India	A
105	Arun	30-Mar-00	India	A
106	Vinutha	30-Mar-00	India	A

Fig 16) Employees table representation

Eid_fk	Work_Country	Effect	Effseq
101	India	30-Sep-12	1
101	United States	03-Jun-19	1
101	Australia	03-Jun-19	2
102	India	31-May-19	1
102	Italy	31-May-19	2
105	India	21-May-19	1
106	United States	01-Jun-19	1

Fig 17) Addresses table representation

A sample sub-query sketch of the form*? h op η* can be represented by a lambda expression, substituted with bound variables, and reduced (by β-reduction) is as below.

(λy.M)[x := N] ⇓ (λy . x op M) N

λ[employee_id].E1[A.employee_id]:=[rex_employees]E1

⇓ λ[employee_id].E1[A.employee_id] op (Select max([E1.id]) [rex_employees]E1)

⇓ λ[employee_id].E1[A.employee_id] = (Select max([E1.id]) [rex_employees]E1)

A.?[employee_id] = (Select max(?[E1.id]) from ?[rex_employees] E1)

Now, replace the pure data structure component, ie table E1 by extending it with functions and values (for the appending the clause - WHERE E1.employee_id = E.employee_id) to build up a return value eventually.

A.?[employee_id] = (Select max(?[E1.id]) from ?[rex_employees] E1 WHERE E1.employee_id = E.employee_id)

Eid_*fk*	work_country	Effdt	Effseq
101	India	30-Sep-12	1
101	United States	03-Jun-19	1
101	Australia	03-Jun-19	2
102	India	31-May-19	1
102	Italy	31-May-19	2
106	United States	43617	1

Fig 18) Query Output with the first Sub-Query

The above sub-query eliminated the row for Emp_id=105, as he did not have a foreign address row. Now, adding Πmax(effdt)(Addresses) returned only the latest effective dated address rows:

Eid_*fk*	Work_Country	Effdt	Effseq
101	Australia	03-Jun-19	2
102	Italy	31-May-19	2
106	United States	01-Jun-19	1

Fig 19) Query Output with the effdt Sub-Query

$\Pi_{g(max(effdt,\ effseq)}(Addresses\ \ _{Eid\text{-}FK}\ \ _{Eid}\ Employees)$ returns only the latest effective sequenced(effseq) rows within the latest effective date rows:

```
SELECT E.employee_id, E.first_name AS Employee_name, A.Country_name as work_country, A.effdt,A.effseq, E.designation, E.empl_rcd
FROM rex_employees E
JOIN rex_departments D ON (E.department_id = D.id)
JOIN rex_locations L ON (D.location_id = L.id)
JOIN rex_countries C ON (L.country_id != C.id)
JOIN rex_addresses A ON (A.country_id = C.id)
WHERE A.employee_id = (Select max(E1.id) from rex_employees E1 WHERE  E1.id = E.id AND E1.eff_status = 'A')
                      AND A.effdt = (Select max(A1.effdt) FROM rex_addresses A1 WHERE A1.employee_id = E.id AND A1.effdt <= now())
                      AND A.effseq = (Select max(A2.effseq) FROM rex_addresses A2 WHERE A2.employee_id = E.id AND A2.effdt = A.effdt)
AND A.eff_status = 'A';
```

Fig 20: The SQL generated

Eid_*fk*	work_country	effdt	Effseq
101	Australia	3-Jun-2019	2
102	Italy	31-May-2019	2
106	United States	1-Jun-2019	1

Fig 21) Query Output with the effdt, effseq Sub-Queries

For the tables of Fig 15 & 16, the final query retrieved only three rows for Emp_ids 101, 102, and 106 which is the desired outcome. The variables in the FROM and WHERE clauses of the Sub-Query "FROM rex_addresses A1 WHERE A1.employee_id = E.id AND A1.eff_status = 'A' AND A1.effdt" and the entire expression can form a fixed point combinator as these variables are closed variables and the entire expression is repetitive (as seen in the next Sub-Query expression with alias name A2).

12. RESULTS

The results can be quickly verified first by executing the synthesized SQL at the same database where it is constructed. The results can now be verified by comparing the expected row count with the actual rows returned by executing the SQL generated. This can be further confirmed by checking all the data values (if returned rows count is in the range of a few hundred) or by looking for the most prominent tuples in the result set produced by running the SQL.

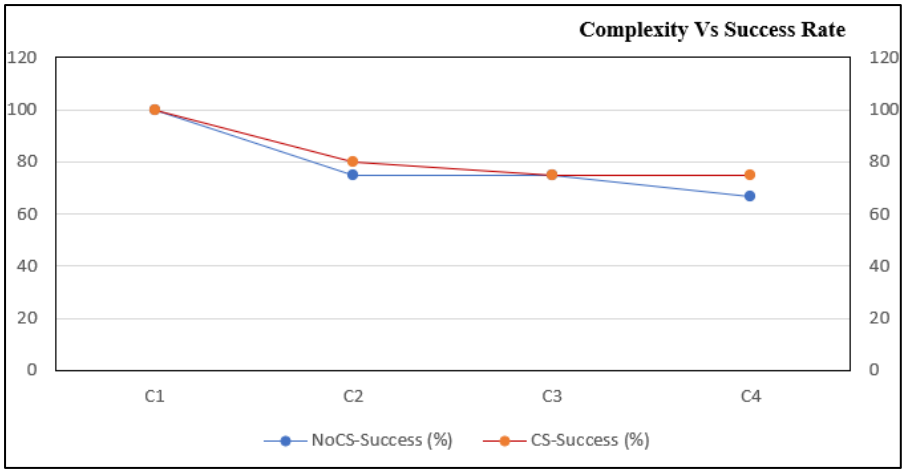


Fig: 22 Success Vs Complexity Graph

The standard measures of question-answering systems – Precision and Recall are used to evaluate the correctness of our system.

Precision: Measures the number of accurate or expected results compared to actual results. Higher rates of precision mean very few false positives (additional rows).

$$\text{Precision} = \frac{\{\text{expected results}\} \cap \{\text{actual results}\}}{\{\text{actual results}\}} \quad \text{OR}$$

$$\text{Precision} = \frac{\{\text{expected rows}\} \cap \{\text{actual rows}\}}{\{\text{actual rows}\}}$$

DB	Complexity	Expected Row Count	Actual Row Count	{E}∩{A}	Precision
REX	C1	24	24	24	1
	C2	20	20	20	1
	C3	20	16	16	0.80
	C4	5	5	5	1
				Average	**0.95**

Table 3: Precision Measure

Recall Measures the number of accurate results compared to expected results. Higher rates of recall mean very few false negatives (or missed rows).

$$\text{Recall} = \frac{\{\text{expected results}\} \cap \{\text{actual results}\}}{\{\text{expected results}\}} \quad \text{OR}$$

$$\text{Recall} = \frac{\{\text{expected rows}\} \cap \{\text{actual rows}\}}{\{\{\text{expected rows}\} \cup \{\text{actual rows}\}\}}$$

DB	Complexity	Expected Row Count	Actual Row Count	{{E}∪{A}}	Recall
REX	C1	24	24	24	1
	C2	20	20	3	1
	C3	20	16	4	0.8
	C4	5	5	5	1
				Average	**0.95**

Table 4: Recall Measure

Dataset	System	Precision	Recall	Portability
Employee	REX	95.00%	95.00%	Customized
Mooney Geography	PANTO	88.05%	85.86%	No Customisation
	Querix	86.08%	87.11%	No Customisation
	NLP-Reduce	70.70%	76.40%	No Customisation
Mooney Restaurants	PANTO	90.87%	96.64%	No Customisation
	NLP-Reduce	67.70%	69.60%	No Customisation
Mooney Jobs	PANTO	86.12%	89.17%	No Customisation
Software Engineering Ontology	QuestIO	82.14%	71.87%	No Customisation
	AquaLog	86.36%	59.37%	No Customisation
Geographical Facts about Germany	ORAKEL	80.60-84.23%	45.15% - 53.7%	Customized
Library Data	E-librarian	97%	NA	NA

Table 5: Compare NLIDB Systems

DB	Complexity	Input Count	SQL #	%
REX	C1	12	12	100
	C2	10	8	80
	C3	4	3	75
	C4	4	3	75
	Total	**36**	**26**	**82.50**

Table 6: Results without Context Switching

DB	Complexity	DRS Count	SQL Count	%
REX	C1	12	12	100
	C2	10	8	80
	C3	4	3	75
	C4	4	3	75
	Total	**36**	**26**	**82.5**

Table 7: Results with Context Switching

The overall success rate of this work stood at 87.5% which is outstanding across the board (from Complexity C1 to C4). In the case of Complex Queries (containing nested Queries), our

initial results show an 82.5% success rate which is highly promising as very few systems had attempted to synthesize SQLs with nested Queries. As the deductive synthesis approach is adopted in this work which primarily operates over intermediate representation corresponding to partial programs which are validated by schema probing, it hardly had to reject invalid candidate programs and backtrack.

Our Initial experiment's results show that by applying Context Switching as a repair technique, Rex's success rate went up by 5% and the overall success rate stood at 87.5% which is a reasonably good improvement across the board. Even without applying Context Switching by combining deduction and schema driven search methods Rex's success rate stood at 82.5% which is also remarkable as very few systems attempted to synthesize SQLs containing Sub-Queries.

12.1 Results Discussion

The experimental dataset design is kept in a relational (PostgreSQL) ERP database with a wholly normalized schema. It consists of collections of entities (E) manifested as tables or view contexts, their properties (P) as table/view columns, and functional relations (RF) representing inheritance (RI) hierarchies and memberships (RM) for flat relations. The Database Configuration of REX is as given below.

Database Type	Database Name	Size	#Tables	#Columns
Postgres	REX	120MB	12	117

Table 8: Experimental Database Instance

Model Name	Operating System	Processing Speed	RAM	Processor
HP 15s-fr2005tu	Windows 10	2.4 GHz	8 GB	Intel i5 - 4 Cores

Table 9: Experimental Server Configuration

Query Type	Input DRS Count	Success	Avg. Time (Without FP Constructs)	Avg. Time (With FP Constructs)
Nested	4	4	475 msecs	315 msecs

Table 10: Results Summary

The SQLs generated by the REX framework were executed against the REX database instance a) first without the Functional Programming constructs b) then with the Functional Programming constructs (using Monad comprehension and Foldr) discussed in this work. By incorporating the Functional Programming constructs the speed of execution improved by 33%.

Program synthesis is accomplished by employing advanced functional programming constructs like Moands and Foldrs capable of operating on recursive data structures, ordering, and integrating its constituent parts, supporting recursive processing along the way, producing tuple return values - making these constructs the most suitable programming paradigms for constructing nested executable statements in a Domain Specific Language like SQL that eventually returns rows of data as tuples.

To be precise, the adoption of advanced functional programming constructs (Monads and Foldrs) facilitated the translation of complex functional requirements with rich domain-specific semantics, offered the needed syntactic sugar, and enabled scalable program synthesis as evident from the above results.

13. CONCLUSIONS

We have proposed a novel database-schema-aware framework by combining a set of techniques for synthesizing SQL queries from Data Requirement Specifications. Our approach used semantic parsing for generating the initial query sketches, which were then completed using a program-directed SQL synthesis and refined further using Repair techniques. One of the unique contributions of this work is the adoption of a typical deductive inference methodology for program synthesis. As the strategy adopted in this work for program synthesis is predominantly based on deductive inference and heuristic search, it could leverage the strengths of deductive, syntax-guided, and domain-specific search approaches in one algorithm. Even though REX is a database Domain-Specific NLIDB System, it can be easily customized to make it domain-independent by changing the Schema Probing SQL typical to each database. Though the query building program builds the query Top to bottom, query Execution engines are typically bottom-up engines where rules (predicate clauses) have actions in the consequence.

We have also suggested the use of Lambda calculus for intermediate representation and transformation Applying Lambda Calculus-based operations (features like entity bifurcation and substitution) greatly aided in making the translation more systematic. Adopting Functional programming techniques like fixed-point combinators permitted the iterative application of query fragments while forming nested intermediate structures. Iterations were performed by transforming relational clauses as parameterized recursive functions. The finding that nested queries have a bearing with nested functions and can be represented and implemented through fixed-point combinators of Lambda calculus is the novelty of this part of the work. Some of the clauses were generated in their applied context, without even applying the operations of Lambda-Calculus. Our experiment shows combining ideas from databases, denotational semantics, functional programming, and NL processing communities is effective in

accomplishing the translation. Our techniques applied through an automation program; are successful in synthesizing nested SQLs from a given DRS targeting an ERP database and is recommended for use in NLIDB systems. The techniques proposed in this work synthesized sub-queries with one level of nesting with ease and proceeded with the synthesis for further levels of nesting. This work accommodates DML Queries (Update, Delete and Insert) with ease by incorporating minor changes in the Selection clause.

14. FUTURE WORK

The system can be allowed in the future to display a set of possible queries for the input DRS at the intermediate level itself so that they can be evaluated either based on their explanations or the possible results displayed by the system on executing them. Hence work on improving the methods to verify if an intermediate logical expression can lead to a well-formed SQL and methods to rectify those can be pursued.

- This work does not accommodate UNION Query creations leaving it for future work.
- Methods for improving the communication mechanism between expressions across nested contexts can be explored.
- To make the query synthesis system more effective, a user interaction module with Speech Recognition can be included for translating vocal inputs along with textual inputs
- This work may also be extended for Data Definition Languages (DDL).

REFERENCES

[1] Yaghmazadeh, Navid, Wang, Yuepeng, Dillig, Isil, and Dillig, Thomas. (2017, Feb). "Type and Content-Driven Synthesis of SQL Queries from Natural Language". Computer Science - Databases, Computer Science - Programming Languages, eid =arXiv:1702.01168. Available: https://arxiv.org/abs/1702.01168.

[2] Grust T. and Scholl M. H. (1999). "How to comprehend queries functionally". J. Intell. Inf. Syst., vol. 12, no. 2, pp. 191–218.

[3] Androutsopoulos I., Ritchie G. D., and P. Thanisch, (1995). "Natural language interfaces to databases – an introduction". *Natural Language Engineering*.

[4] Bhalotia G., Hulgeri A., Nakhe C., Chakrabarti S.and Sudarshan S. (2002). "Keyword searching and browsing in databases using BANKS". In *ICDE*.

[5] Le V., Gulwani S., and Su Z. (2013). "Smartsynth: Synthesizing smartphone automation scripts from natural language". In MobiSys, pages 193-206.

[6] Carpenter B. (1997). "*Type-logical semantics"*.MIT press.

[7] Cheung A., Solar-Lezama A. and Madden S., (2013), “Optimizing database-backed applications with query synthesis”. In PLDI, pages 3–14.

[8] Desai A., Gulwani S., Hingorani V., Jain N., Karkare A., Marron M., Roy S. R. and Roy S. (2016), “Program synthesis using natural language”. In *ICSE*.

[9] Emani K. V., Ramachandra K., Bhattacharya S. and Sudarshan S. (2016). “Extracting equivalent SQL from Imperative code in database applications,” Proc. ACM SIGMOD Int. Conf. Manag.Data, vol. 26-June-2016, pp.1781–1796.

[10] Wiedermann B., Ibrahim A., and Cook W. R. (2008), “Inter-procedural query extraction for transparent persistence,” Proc. Conf. Object-Oriented Program. Syst. Lang. Appl. OOPSLA, pp.19–36.

[11] Tran Q. T., Chan C., and Parthasarathy S. (2009). “Query by Output”. In SIGMOD, pages 535-548.

[12] Zloof M. M. (1975). “Query-by-example: the invocation and definition of tables and forms”. In VLDB.

[12b] M. M. Zloof, "Query-by-Example: A database language," in *IBM Systems Journal*, vol. 16, no. 4, pp. 324-343, 1977, DOI: 10.1147/sj.164.0324.

[13] Gulwani S. and Marron M. (2014). “Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation”. In *SIGMOD*.

[14] Hristidis V. and Papakonstantinou Y. (2003).“DISCOVER keyword search in relational databases. In V*LDB*.

[15] Jose M. and Majumdar R (2011). “Cause clue clauses: error localization using maximum satisfiability”. *PLDI*.

[16] Kate R. J. and Mooney R. J. (2006). “Using string kernels for learning semantic parsers”. In *ACL*.

[17] Talpin, Jean-pierre and Jouvelot, Pierre (1992). “The Type and effect Discipline”. In Information and Computation, pages 162-173.

[18] Zhang S. and Sun Y. (2013). “Automatically synthesizing SQL queries from input-output examples”.In *ASE*.

[19] Li F. and Jagadish H. V. (2014). “Nalir: an interactive natural language interface for querying relational databases”. In *VLDB*, pages 709–712.

[20] Li Y., Yang H. and Jagadish H. V. (2006). “Constructing a generic natural language interface for an XML database”. In *EDBT*, pages 737–754.

[21] Liang P. and Potts C. (2015). “Bringing machine learning and compositional semantics together”. *Annual Review of Linguistics*, 1(1):355–376.

[22] MacCartney B. and Manning C. D. (2009). “An extended model of natural logic”. In *IWCS*, pages 140–156.

[23] Mikolov T., Sutskever I., Chen K., Corrado G. S., and Dean J. (2013). “Distributed representations of words and phrases and their compositionality”. In *NIPS*.

[24] Hosu I., Iacob R., Brad F., Ruseti, S. and Rebedea T. (2018). “Natural Language Interface for Databases Using a Dual-Encoder Model”. Proc. 27th Int. Conf. Comput. Linguist., pp. 514–524.

[25] Grust T., Kröger J., Gluche D., Heuer, A. and Scholl M. H. (1997). “Query evaluation in CROQUE: Calculus and algebra coincide,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 1271, pp.84–100.

[26] Popescu A., Armanasu A., Etzioni O., Ko D. and Yates A. (2004). “Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability”. In *COLING*.

[27] Popescu A., Etzioni O. and Kautz H. A. (2003). “Towards a theory of natural language interfaces to databases”. In *IUI*, pages 149–157.

[28] Quirk C., Mooney R., and Galley M. (2015). “Language to code: Learning semantic parsers for if-this-then-that recipes”. In *ACL*, pages 878–888.

[29] Raza M., Gulwani S. and Milic-Frayling N. (2015). “Compositional program synthesis from natural language and examples”. In *IJCAI*

[30] Simitsis A., Koutrika G., and Ioannidis Y. (2008). “Précis: From unstructured keywords as queries to structured databases as answers”. *VLDB*

[31] Lezama S.A. (2008). “*Program Synthesis by Sketching”*. Ph.D. thesis, EECS Department, University of California, Berkeley

[32] Grosz B. J., Appelt D. E., Martin P. A. and Pereira F. C. N. (1987). “TEAM: an experiment in the design of transportable natural language interfaces”. *Artificial Intelligence*, 32(2):173–243.

[33] Zelle J. M. and Mooney R. J. (1993). “Learning semantic grammars with constructive inductive logic programming”. In AAAI, pages 817–822

[34] Bognar M., “Contexts in Lambda Calculus”. *Vrije Univ. Amsterdam*, Thesis, pp. 1–236, 2002.

[35] Schüle, M. E., Vorona, D., Lang, H., Kemper, A., Günnemann, S., & Neumann, T. (2019). The Power of SQL Lambda Functions, 534–537.

[36] Fowler, Martin (1999). “Refactoring: Improving the Design of Existing Code”. Addison-Wesley. pp. 63ff. ISBN 978-0-201-48567-7.

[37] Cooper, E. (2009). “The script-writer’s dream: How to write great SQL in your language, and be sure it will succeed”.Springer-Verlag Berlin, Heidelberg, (DBPL), pp 36-51.

[38] Selinger, P. (2013). “Lecture Notes on the Lambda Calculus”. Department of Mathematics and Statistics. Dalhousie University, Halifax, Canada.

[39] Grossman, D., Morrisett, G., & Zdancewic, S. (2000). “Syntactic type abstraction”. ACM Transactions on Programming Languages and Systems, 22(6), 1037–1080. https://doi.org/10.1145/371880.371887

[40] Agrawal S., Chaudhuri S., and Das G. (2002). “DBxplorer: A system for keyword-based search over relational databases”. In ICDE, pages 5–16.

[41] Lavender G. (1936). “Introduction to the λ-Calculus Part ICS 209 - Functional Programming Alonzo Church, The λ -calculus in Computer Science,” no. 1930, pp. 1–16.

[42] Van Leeuwen P. (2018), “λ-Calculus Syntax’s Definition and Completeness As Graph Database Querying Language.pdf.” Semantic Scholar, pp. 1–8.

[43] Grust T., Mayr M., Rittinger J. and T. Schreiber. (2009). “FERRY: Database-supported program execution”.SIGMOD-PODS’09 – Proc. Int. Conf. Manag. Data 28th Symp. Princ. Database Syst., pp. 1063–1065.

[44] Bussche, Jan Van den. (2001). “Simulation of the nested Relational algebra by the flat relational algebra, with an application to the complexity of evaluating power set algebra expressions”. Theoretical Computer Science, 254(1-2):363- 377.

[45] V. Breazu-Tannen, P. Buneman, and S. Naqvi, “Structural recursion as a query language,” DBPL3 Proc. third Int. Work. Database Program. Lang. bulk types persistent data, no. August, pp. 9–19, 1992.

[46] M. Bognar, “Contexts in Lambda Calculus,” Vrije Univ. Amsterdam, Thesis, pp. 1–236, 2002.

[47] Buneman P., Fernandez M., and Suciu D., “UnQL: A Query language and algebra for semistructured data based on structural recursion,” VLDB J., vol. 9, no. 1, pp. 76–10, 2000.

[48] Bird, R.S. (1987). An Introduction to the Theory of Lists. In M. Broy (Ed.), Logic of Programming and Calculi of Design. NATO ASI Series. Springer Verlag, vol. 36, pp. 5–42.

[49] Wadler P., “Comprehending Monads,” Math. Struct. Comput. Sci., vol. 2, no. June 1990, pp. 1–38, 1970.

[50] Gill A., Launchbury J., and Jones S. L. P., “Short cut to deforestation,” no. Section 4, pp. 223–232, 1993.

[51] Buneman P., Libkin, L., Suciu D., Tannen V., and Wong L. (1994). “Comprehension Syntax”, ACM SIGMOD Record.

[52] Grust T., Kröger J., Gluche D., Heuer A. and Scholl M.H., “Query evaluation in CROQUE: Calculus and algebra coincide,” Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 1271, pp. 84–100, 1997.

[53] D. Saha, N. Gantayat, S. Mani, and B. Mitchell. “Natural Language Querying in SAP-ERP Platform,” In Proc. 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany’, 09, 2017, pp 878-883.

D. Saha, A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal, and F. Ozcan, “ATHENA: An Ontology-Driven System for Natural Language Querying over Relational Data Stores,” In Proc. of the VLDB Endowment (PVLDB), IBM Almaden Research Center, California, vol. 9, no. 12, 2016, pp. 1209–1220

[54] Jan Van den Bussche. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. Theoretical Computer Science, 254 (1-2):363-377, 2001.

[55] Gulwani, Sumit, “Dimensions in Program Synthesis”, PPDP'10 - Proceedings of the 2010 Symposium on Principles and Practice of Declarative Programming. 13-24, 2010, 10.1145/1836089.1836091.

[56] Poulovassilis, A. and King, P. (1990). Extending the functional data model to computational completeness. Proc. of the 2nd Int’l Conference on Extending Database Technology (EDBT), number 416 in Lecture Notes in Computer Science (LNCS) (pp. 75–91). Venice, Italy: Springer Verlag.

[57] Poulovassilis, A. and Small, C. (1996). Algebraic Query Optimisation for Database Programming Languages. The VLDB Journal, 5(2), 119–132.

[58] F. Li, “Querying RDBMS Using Natural Language,” A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Computer Science and Engineering) in the University of Michigan, 2017, pp 1-87.

[59] P. Pasupat and P. Lang, “Compositional semantic parsing on semi-structured tables”, In Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers, 2015, pages 1470–1480.

[60] Dong L. and Lapata M, “Coarse-to-fine decoding for neural semantic parsing,” ACL 2018 - 56th Annu. Meet. Assoc. Comput. Linguist. Proc. Conf. (Long Pap., vol. 1, pp. 731–742), 2018.

[61] Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, Dragomir R. Radev. "TypeSQL: Knowledge-based Type-Aware Neural Text-to-SQL Generation", NAACL- 2018, Volume2 (Short Papers),588-594.

[62] Krivine, J., J. Roger Hindley and Jonathan P. Seldin. “Introduction to combinators and λ-calculus”. London Mathematical Society student texts, no. 1. Cambridge University Press, Cambridge, etc. 1986, vi 360 pp. Journal of Symbolic Logic, 53(3), 985-986. doi:10.2307/2274588. https://projecteuclid.org/euclid.jsl/1183742736

[63] Cimiano, P., Haase, P., Heizmann, J., & Mantel, M. Orakel: A portable natural language interface to knowledge bases. Data & Knowledge Engineering, 65(2), 325–354. 2008.

[64] Abhijeet Gupta, “Complex Aggregates in Natural Language Interface To Databases”, International Institute of Information Technology, Hyderabad, 2013.

[65] Anthony Klug. “Equivalence of relational algebra and relational calculus query languages having aggregate functions”, J. ACM, pages 699–717, 1982.

[66] Hidaka S., Asada K., Hu Z., Kato H., and Nakano K., “Structural recursion for querying ordered graphs,” ACM SIGPLAN Not., vol. 48, no. 9, pp. 305–318, 2013.

[67] Leonardo Lesmo, “Natural Language Query interpretation in restricted domains”, In the proceeding of 6th International Conference on NLP ICON-08, Pune, India, 2008.

[68] George M and Rohini V, “A novel Framework for Synthesizing Nested Queries in SQL from Business Requirements Language”, Turkish Journal of Computer and Mathematics Education (TURCOMAT), Vol. 12 No. 2 p2680-2691, 2021, https://doi.org/10.17762/turcomat.v12i2.2291.

[69] L.Wong. Querying Nested Collections. Ph.D. thesis, Philadelphia, PA, USA, 1994.

[70] E. L. Robertson, L. V. Saxton, D. V. Gucht, and S. Vansummere, “Structural recursion as a query language on lists and ordered trees”, Theory of Computing Systems, 4.

[71] Alessandra Giordani, "Mapping Natural language into SQL in a NLlDB", Springer, 2008,Yol 5039,pp. 367-371, DOl-I 0.1 007/978-3-540-69858-6 46.

[72] Lin D., "Automatic Retrieval and Clustering of Similar Words", Conference Proceedings-COLING 1998 Volume 2: The 17th International Conference on Computational Linguistics, 1998, https://aclanthology.org/C98-2122.

[73] S. Bergamaschi, E. Domnori, F. Guerra, R. T. Lado, and Y. Velegrakis, “Keyword search over relational databases: a metadata approach”. In SIGMOD, pages 565–576. ACM, 2011.

[74] S. Bergamaschi et al, “QUEST: A Keyword Search System for Relational Data Based on Semantic and Machine Learning Techniques”, Proc. VLDB Endow., pages 1222–1225, 2013.

[75] X. Yang, C. M. Procopiuc, and D. Srivastava, “Summary graphs for relational database schemas”. PVLDB, 4(11):899–910, 2011.

[76] Gray, P.M.D., Kerschberg, L., King, P.J.H., Poulovassilis, A.: “The Functional Approach to Data Management”. Springer, Berlin (2004).

[77] Barker, “Case Method: Entity-Relationship Modeling”. Addison-Wesley Publ. Comp., Boston (1990).

[78] Pokorný, J., “A function: unifying mechanism for entity-oriented database models”. In: Batini, C. (ed.) Entity-Relationship Approach, pp. 165–181, Elsevier Science Publishers B.V, North-Holland, 1989.

List of Tables

Sequence	Description	Page No
Table 1	Comparative list of top SQL Automations	7
Table 2	SQLs by Complexity	7
Table 3	Precision Measure	32
Table 4	Recall Measure	33
Table 5	Compare NLIDB Systems	33
Table 6	Results without Context Switching	33
Table 7	Results with Context Switching	33
Table 8	Experimental Database Instance	34
Table 9	Experimental Server Configuration	34
Table 10	Results Summary	34

List of Figures

Sequence	Description	Page No
Fig 1	System Components	2
Fig 2	Framework Diagram	8
Fig 3	Database Domain Ontology	9
Fig 4	Schematic overview of the approach	10
Fig 5	High-Level Overview of the System	12
Fig 6	Sample Output of the Schema Probing SQL	13
Fig 7	Transformation Flow	14
Fig 8	A Natural Language Query (NLQ) annotated for Ontology Mapping	15
Fig 9	A Graph to Tree Conversion	16
Fig 10	Graphical depiction of the top-ranked interpretations of the NLQ given in Fig 8	17
Fig 11	Grammar of Extended Relational Algebra	18
Fig 12	Lambda Expression Representation	23
Fig 13	Simplified schema of an HRMS ERP database	26
Fig 14	Representation using Relational Algebra	26
Fig 15	DB Table Relationship	29
Fig 16	Employees table representation	30
Fig 17	Addresses table representation	30
Fig 18	Query Output with the first Sub-Query	30
Fig 19	Query Output with the effdt Sub-Query	31
Fig 20	The SQL generated	31
Fig 21	Query Output with the effdt, effseq Sub-Queries	31
Fig 22	Success Vs Complexity Graph	32

Conferences and Publications

Journal / Conf Name	Type	Status	Date	Paper
ICSAC	Conference Paper	Published in Conference Proceedings	02-Mar-2018	“Natural Language Processing: State-of-the-Art and Deliberations for Optimization.” Paper ID: P1802098
TURCOMAT	Journal Paper	Published	01-Jun-2021	"A novel framework for synthesizing nested queries in SQL from business requirements language". https://doi.org/10.17762/turcomat.v12i2.2291
Malaysian Journal of Computer Science	Journal Paper	Submitted	06-Nov-2021	“Context Switching in Repairing nested SQL Queries”.
[IJDTA] International Journal of Database Theory and Application	Journal Paper	Submitted	01-Nov-2021	“Synthesizing SQLs from Natural Language Statements using Lambda Calculus”.